Université Assane Seck de Ziguinchor

UFR Sciences et Technologies

Département Informatique

MEMOIRE DE FIN D'ÉTUDES

Pour l'obtention du diplôme de Master

Mention : Informatique

Spécialité : Génie Logiciel

Sujet :

Automatisation du système de restauration de l'Université Assane Seck de Ziguinchor, de la vente des tickets au contrôle des accès aux restaurants.

Présenté par :

M. Babacar DIAGO

Soutenance le 04/02/2022

Membres du jury

- Pr. Youssou FAYE (**Président du jury**)
- M. Bassirou DIENE (**Rapporteur**)
- Pr. Youssou DIENG (**Examinateur**)
- Dr. Ibrahima DIOP (**Encadreur**)

Sous la direction de :

- Dr. Ibrahima DIOP

Sous la supervision de :

- Pr. Youssou FAYE

Année Universitaire : 2020 – 2021

Résumé

Le Centre des Œuvres Universitaires de Dakar et les Centres Régionaux des Œuvres Universitaires et Sociales de Ziguinchor, Bambey, Saint-Louis et Thiès occupent une place très importante au sein des Universités. Ils sont chargés de la gestion des œuvres sociales destinées aux étudiants afin de leur offrir de meilleures conditions de vie. Leurs services de restauration est indispensable dans la vie quotidienne de l'étudiant.

Ainsi l'augmentation considérable du nombre d'étudiants, a entrainé une complexité dans la gestion des activités du service de restauration particulièrement celle des tickets et de l'accès au restaurant universitaire.

Pour mener à bien leurs missions, ces services de restauration des différents centres des Œuvres Universitaires doivent avoir un système informatique pour la gestion de leurs activités quotidiennes.

Cela nous a conduits à proposer, dans le cadre de ce mémoire de fin d'étude de master, une application basée sur les microservices pour la gestion des tickets et de l'accès au restaurant universitaire. Cette application permet l'automatisation de la gestion de tickets et d'accès au restaurant de l'Université. Elle consiste à dématérialiser les tickets de restauration et permettre aux étudiants de pouvoir acheté des tickets sans se déplacer grasse à la porte money Univ-Money. Chaque étudiant est identifié par un codeQR. L'application permettra aussi aux contrôleurs d'accès au restaurant universitaire d'avoir un système de contrôle d'accès qui leurs permettra d'identifier les étudiants grâce à un lecteur QR et de leurs débiter un ticket.

Pour une bonne gestion de ce projet, nous avons opté pour la méthode (ou processus) unifiée **2TUP**. Toujours dans le cadre du projet, nous avons utilisé l'architecture microservice. Pour l'implémentation nous avons utilisé les frameworks spring boot coté back-end et Angular coté front-end. Nous avons utilisé le Système de Gestion de Bases de Données Relationnelles (SGBDR) MySQL pour le stockage des données.

Mots clés : **Gestion des tickets, Gestion des accès, Restaurant Universitaire**

Abstract

The Centre des Œuvres Universitaires de Dakar and the Centres Régionaux des Œuvres Universitaires et Sociales of Ziguinchor, Bambey, Saint-Louis and Thiès occupy a very important place within the universities. They are in charge of managing social works for students in order to provide them with better living conditions. Their catering services are essential in the daily life of the student.

The considerable increase in the number of students has led to a complexity in the management of the activities of the catering service, particularly that of tickets and access to the university restaurant.

In order to carry out their missions, the catering services of the various university welfare centres must have a computer system to manage their daily activities.

This led us to propose, within the framework of this Master's thesis, a microservices-based application for managing tickets and access to the university restaurant. This application allows the automation of the management of tickets and access to the university restaurant. It consists in dematerialising the restaurant tickets and allowing students to buy tickets without going to the Univ-Money door. Each student is identified by a QR code. The application will also allow the access controllers at the university restaurant to have an access control system that will allow them to identify students through a QR reader and to debit them a ticket.

For a good management of this project, we opted for the 2TUP unified method (or process). For the project we used the microservice architecture. For the implementation we used the spring boot framework on the back-end and Angular on the front-end. We used the MySQL Relational Database Management System (RDBMS) for data storage.

Keywords: Ticket management, management access, University restaurant

Remerciements

Je rends grâce à ALLAH, le Miséricordieux, le tout Miséricordieux.

Je tiens à exprimer mes très sincères remerciements à mon encadrant Dr. Ibrahima DIOP pour sa disponibilité, ses conseils et ses encouragements qui m'ont permis de réaliser ce travail dans les meilleures conditions.

Nous remercions Pr. Youssou FAYE d'avoir accepté de présider notre jury ainsi que les autres membres de ce jury - Pr. Youssou DIENG et M. Bassirou DIENE - d'avoir accepté d'évaluer ce modeste travail.

J'adresse aussi mes reconnaissances à tous les enseignants du département d'Informatique de l'UASZ, pour la richesse et la qualité de leurs enseignements et les efforts fournis pour assurer à leurs étudiants une bonne formation.

Je remercie tout le personnel du CROUS-Z particulièrement ceux qui m'ont accordé leur temps pour la collecte des informations.

Je remercie mes camarades de classe pour leur collaboration tout au long de ma formation.

Je profite de cette occasion pour rendre hommage à :

> *Feue Mamour DIOUF et Feue Alpha SANE, se sont des camarades de classe qui ont rendu l'âme cette année. Que Dieu les accueils dans son paradis !*

Je remercie mon tuteur Elhadji Saliou DIAGO petit frère de mon père et à ses épouses qui m'ont toujours soutenu pendant toutes les années que j'ai passé à Ziguinchor.

Enfin je tiens à dire combien le soutien quotidien de ma famille a été important toutes les années de ma formation, je leur dois beaucoup.

Dédicaces

A ma Mère, Rokhy BITEYE, en vous je vois la maman parfaite, toujours prête à se sacrifier pour le bonheur de ses enfants.

A mon Père, Abdoulaye DIAGO, en vous je vois un père dévoué à sa famille. Ta présence en toute circonstance m'a plusieurs fois rappelé le sens de la responsabilité.

Table des matières

Liste des tableaux

Liste des figures

Liste des abréviations

UASZ : Université Assane SECK de Ziguinchor

API : Application Programming Interface

JPA : Java Persistence API

REST : REpresentational State Transfer

UML : Unified Modeling Language

2TUP : 2 Track Unified Process

URI : Uniform Ressource Identifier

RU : Restaurant Universitaire

CROUS-Z : Centre Régional des Œuvres Universitaires et Sociales de Ziguinchor

DES : Direction du Service aux Etudiants

XP : eXtreme Programming

RUP : Rational Unified Process

MLD : Model Logique des Données

IDE : Environnement de Développement Intégré

SGBD : Système de Gestion de Base de Données

JDBC : Java DataBase Connectivity

POJO : Plain Old Java Object

CRUD : Create/Read/Update/Delete

INTRODUCTION GENERALE

L'informatique connait une avancée technologique considérable dans tous les secteurs d'activité. Elle y est présente et est indispensable pour leur bon fonctionnement. En effet, elle facilite le travail du personnel, assure la rapidité et l'efficacité des taches.

Par conséquent toutes structures qui se veut moderne doit penser à avoir un système d'information qui lui permettra de stocker ses données et de les traiter. C'est le cas du Centres Régional des Œuvres Universitaires et Sociales de Ziguinchor (CROUS-Z) qui gère plusieurs services dont le service de restauration qui occupent une partie importante dans le campus social. Ainsi il rencontre plusieurs difficultés dans la gestion de la restauration universitaire due à son mode de fonctionnement manuel, le contrôle des tickets lent et pénible, le risque de détournement financière, etc.
A cela s'ajoute, le nombre d'étudiant qui augmente par millier chaque année et qui fait qu'il est pertinent d'automatiser le système de restauration de l'université Assane Seck de Ziguinchor, de la vente des tickets au contrôle des accès aux restaurants universitaire.

Cette automatisation consiste à dématérialiser les tickets de restauration et permettre aux étudiants de pouvoir acheté des tickets sans se déplacer grasse aux portes money existant comme Orange Money, Free Money, Wave et Univ-Money. Elle permettra aussi aux contrôleurs d'accès au restaurant universitaire d'avoir un système de contrôle d'accès qui leurs permettra d'identifier les étudiants et de leurs débiter un ticket.

C'est dans ce cadre que se situe ce présent mémoire. Il comprend cinq (05) chapitres :

Le premier chapitre décrit le sujet du mémoire. En effet, il permet d'exposer les problèmes que rencontre le service de restauration et de dégager la problématique du sujet. Ce premier chapitre permet aussi de fixer les objectifs de ce présent mémoire.

Le deuxième chapitre intitulé spécification et analyse des besoins fonctionnels permet de faire une présentation des acteurs et des fonctionnalités du système de « gestion des tickets et des accès au restaurant universitaire ». Il intègre les diagrammes de cas d'utilisation montrant les relations entre les acteurs et les fonctionnalités du système. Il permet aussi de faire l'analyse des besoins fonctionnels du système en utilisant des diagrammes d'activité et de séquence.

Le troisième chapitre porte sur la conception du système. Il aborde l'ensemble des diagrammes de composants, de package et de déploiement, mais aussi la conception détaillée illustrant les diagrammes de classes et du dictionnaire de données nécessaire pour la réalisation du projet.

Le quatrième chapitre dénommé implémentation du système illustre le codage de l'application, mais aussi les outils de développement utilisés pour l'implémentation.

Le cinquième et dernier chapitre dénommé présentation de l'application est composé d'un ensemble de captures présentant l'application.

CHAPITRE I : DESCRIPTION DU SUJET

Ce premier chapitre porte sur la description du sujet. Il aborde la présentation du Centre Régional des Œuvres Universitaire et Sociales de Ziguinchor (CROUS-Z) et du **service de restauration** de l'Université Assane Seck de Ziguinchor. Il décrit aussi les problèmes rencontrés dans la gestion de ce dernier, aborde la problématique du sujet et d'exposer les objectifs du projet.

I.1 Etude de l'existant

Cette partie consiste à étudier ce qui existe déjà au niveau du **service de restauration** de l'Université Assane Seck de Ziguinchor.

I.1.1 Présentation du Centre Régional des Œuvres Universitaires Sociales de Ziguinchor (CROUS-Z)

Placée sous la tutelle du Rectorat, la direction du service aux étudiants (DSE) était chargée de la gestion des œuvres sociales destinées aux étudiants. Cette direction était chargée de l'accompagnement afin d'offrir de meilleures conditions de vie aux étudiants.

Cette DSE est remplacée par le Centre Régional des Œuvres Universitaires Sociales de Ziguinchor qui est installé le 13 novembre 2017. Adoptée par l'Assemblée Nationale en sa séance du vendredi 19 février 2016, cette loi a été introduite dans un contexte d'élargissement de la carte universitaire pour offrir aux étudiants un meilleur cadre de vie, améliorer la gouvernance et la gestion des œuvres sociales universitaires. Le CROUS-Z a pour mission d'améliorer les conditions de vie des étudiants.

Il s'occupe de :

- l'hébergement des étudiants ;
- la restauration ;
- la prise en charge médicale ;
- l'animation sociale, culturelle et sportive du campus ;
- des affaires estudiantines (ressources financières et service à la communauté, etc.).
- etc.

Il est composé de plusieurs services :

- le service du logement ;

- le service de la restauration ;
- le service médical ;
- le service de l'animation sociale, culturelle et sportive

I.1.2 Présentation du restaurant universitaire

Un restaurant universitaire (RU) est une cantine, un lieu de restauration collective destiné aux étudiants. Les restaurants universitaires ont pour objectif de servir un repas complet à prix subventionné. Grâce aux restaurants universitaires, les étudiants issus de tous les milieux ont la possibilité de prendre à l'extérieur de chez eux les repas du jour. Ce restaurant fonctionne tous les jours pendant l'année universitaire à des heures bien précises pour le petit déjeuner, le déjeuner et le diner. Il faut des tickets pour accéder au restaurant ; ces derniers sont vendus à 50 francs pour le petit déjeuner et 100 francs pour les déjeuner et diner.

Le restaurant universitaire est géré par un repreneur privé à qui, le directeur du CROUS-Z confie une mission de servir des repas aux étudiants, sur la base d'un cahier de charges et d'un appel d'offres fait sur la base du code des marchés publics. Il fonctionne par le biais d'un formatage c'est-à-dire que le CROUS-Z met à la disposition du gérant tout le matériel nécessaire. Ce dernier (le repreneur) l'exploite pour le compte du CROUS-Z moyennant une contrepartie.

Ainsi, un contrat est signé pour que le CROUS-Z verse un montant tous les 10 jours dans le compte du gérant. Ce repreneur est accompagné d'un personnel déjà en place pour assurer le bon fonctionnement du restaurant. Ce personnel appartenant au CROUS-Z est composé de :

- un chef de cuisine ou chef cuisine qui élabore l'ensemble des plats proposés à la carte du restaurant tout en coordonnant l'activité de la cuisine avec celle de la salle pour assurer le service. Il veille aussi au respect des normes de sécurité et d'hygiène dans sa salle de cuisine ;
- un maitre d'hôtel qui est chargé de la coordination de l'ensemble du personnel de service, dont le service de table ;
- des contrôleurs, généralement deux, dont le premier représentant le repreneur se charge de la facturation des tickets et le second mandaté par le CROUS-Z fait un contrôle sur les tickets ;

Cependant, malgré les efforts qu'ils effectuent à l'endroit des étudiants, nous remarquons qu'ils rencontrent d'énormes problèmes dans sa gestion particulièrement dans la vente de

tickets, mais aussi l'accès au restaurant. Nous allons relater dans la partie qui suit, les processus de la vente de ticket et du contrôle d'accès au restaurant et les difficultés rencontrées dans le mode de fonctionnement du restaurant.

I.1.3 Les processus de la vente de ticket et du contrôle d'accès au restaurant universitaire

Dans cette partie nous allons décrire l'ensemble des étapes qui constitue la vente de tickets et le contrôle d'accès au restaurant.

Vente de tickets

Le caissier (vendeur de ticket) s'approvisionne en tickets à la direction du CROUS-Z. Lors de l'approvisionnement, une fiche est remplie et signé par le caissier et le chef bureau recouvrement (voir **annexe 1**).

Cette fiche contient les informations suivantes :

- la date de l'approvisionnement,
- le numéro d'approvisionnement,
- les numéros de série contenu par chaque brochure
- le type de ticket (petit déjeuner ou déjeuner)
- le nombre de ticket
- la valeur totale en CFA

Après l'approvisionnement, la prochaine étape revient à vendre les tickets reçus aux étudiants.

A la fin de la journée, le caissier devra faire un versement chez le caissier général à la direction en lui remettant une fiche renfermant l'ensemble des informations nécessaires pour le versement. Cette fiche est signée par le caissier qui fait le versement, le caissier général et le chef du bureau recouvrement. La fiche contient les informations suivantes (voir **annexe 2**) :

- la date du versement
- le numéro du versement
- les numéros de série des tickets vendus
- le type de ticket (petit déjeuner ou déjeuner)
- le nombre de ticket
- et le montant total du versement en Francs CFA

Contrôle de l'accès au restaurant

Parmi les personnelles de la gestion du restaurant universitaire, nous avions cité les contrôleurs, généralement deux, dont le premier représentant le privé se charge de la facturation des tickets et le second mandaté par le CROUS-Z fait un contrôle sur les tickets.

Avant d'accéder aux services offerts par le restaurant, l'étudiant devra se présenter avec sa carte d'étudiant et un ticket correspondant au repas (ticket de petit déjeuné ou de repas).

Chaque ticket contient trois parties dont une pour la direction du CROUS-Z, l'autre pour le contrôle et la dernière partie pour la facturation. Toutes ces partitions servent de contrôle pour la bonne gestion des accès au restaurant.
Les **figures 1** et **2** ci-dessous représentent respectivement un ticket de petit déjeuner et un ticket de repas.

Figure 1 : Ticket de petit déjeuner

Figure 2 : Ticket de repas

Les contrôleurs se chargent de classer les tickets selon les partitions. Les tickets sont paquetés par lot de 20 tickets.

Pour le rapport, ils doivent remplir une fiche de contrôle appelé *fiche de contrôle d'accès aux restaurants universitaires*. Cette fiche est signée par l'ensemble des contrôleurs présent, le gérant et le superviseur (voir **annexe 3**).

La fiche contient les informations suivantes :

- la date,
- le nom du restaurant,
- le type du repas,
- le nombre de lots de 20 de tickets,
- le nombre total de tickets par unité,
- le prix unitaire et la somme totale en Francs CFA.

I.2 Les limites de l'existant

La restauration occupe une place importante et indispensable au sein du CROUS-Z. Malgré cela, ce système rencontre d'énormes problèmes dans la vente des tickets et l'accès au restaurant à cause de son mode de fonctionnement principalement manuel.

Au cours du processus cité précédemment nous avons identifiés plusieurs problèmes tels que :

- les problèmes liés à la gestion des tickets sont :
  - les tickets sont en papier et en couleur, par conséquent, l'impression de ces tickets et la logistique nécessite beaucoup de ressources (machine, papier, encre et personnel).
  - de l'approvisionnement en tickets à leur vente, plusieurs problèmes peuvent être rencontrés comme le risque de perdre des tickets, le risque de faire des erreurs de décompte lors de l'approvisionnement ou lors des ventes (par exemple lorsqu'on donne à un étudiant un nombre de ticket supérieur au nombre qu'on lui devait lors de l'achat) et le risque de perdre l'argent du versement.
  - des longues files d'attente pour acheter des tickets
  - les étudiants constatent des vols de tickets et le risque de les perdre en les abimant.
- les problèmes liés aux contrôles sont :
  - Le risque de ne pas remonter des rapports de versement, d'approvisionnement et d'accès au restaurant vers les autorités.
  - les personnels rencontrent plusieurs difficultés d'établir des bilans journaliers.
  - le problème de disponibilité des données et des statistiques sur l'accès des étudiants au restaurant Universitaire pour aider l'autorité dans la décision
- Etc.

La gestion des tickets et du contrôle des accès au restaurant universitaire rencontre plusieurs difficultés auxquelles nous essayerons d'apporter des solutions dans la partie qui suit.

I.3 Les objectifs de notre projet

Après avoir identifié les difficultés dans la procédure actuelle de gestion du restaurant universitaire particulièrement sur la gestion des tickets et des accès au restaurant, notre solution consiste à concevoir et implémenter un système de dématérialisation complet du système actuel qui corrigera les manquements et les défaillances notées. En effet il consiste à dématérialiser les tickets de restauration et permettre aux étudiants de pouvoir acheté des tickets sans se déplacer grasse à la porte money Univ-Money. Chaque étudiant est identifié par un codeQR. L'application permettra aussi aux contrôleurs d'accès au restaurant universitaire d'avoir un système de contrôle d'accès qui leurs permettra d'identifier les étudiants grâce à un lecteur QR et de leurs débiter un ticket.

Les objectifs majeurs de ce projet sont :

- la dématérialisation des tickets,
- la possibilité d'acheter des tickets en ligne grâce aux portemonnaies électronique
- la dématérialisation du processus de l'approvisionnement des tickets,
- l'automatisation des ventes de tickets,
- la dématérialisation de la procédure de contrôle des accès au restaurant universitaire,
- l'automatisation des rapports,
- la suppression des coups de la gestion des tickets.

Pour une meilleure réalisation du projet, nous avons opté l'architecture microservice qui consiste à découper une application en petits services autonomes.

I.4 Les microservices

Les microservices désignent à la fois une architecture et une approche moderne du logiciel dans laquelle le code d'application est livré en petits morceaux gérables, indépendants les uns des autres.

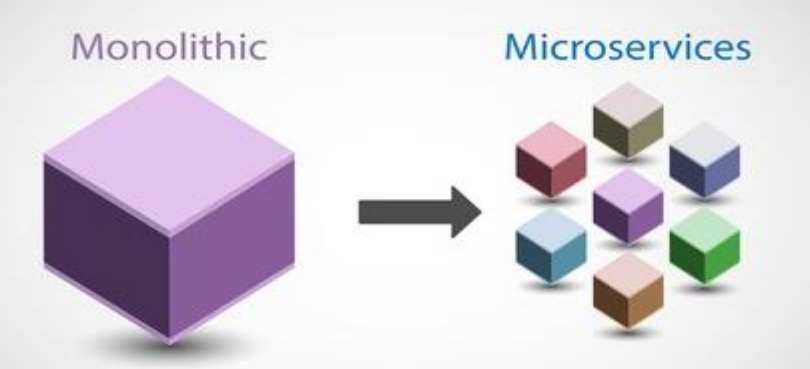


Figure 3 : Découpage d'une application en petits services

L'architecture Microservices propose une solution en principe simple : découper une application en petits services, appelés Microservices, parfaitement autonomes qui exposent une API (Application Programming Interface) que les autres Microservices pourront consommer alors qu'une architecture monolithe est une application dont l'ensemble du code et des fonctionnalités est implémenté dans un seul programme.

Leur petite échelle et leur isolement relatif peuvent entraîner de nombreux avantages supplémentaires, tels que :

- Les microservices définissent des API qui exposent leurs fonctionnalités à n'importe quel client. Les clients pourraient même être d'autres applications.
- Pour communiquer entre eux, les microservices d'une application utilisent le modèle de communication requête-réponse. L'implémentation typique utilise des appels API REST basés sur le protocole HTTP.
- Chaque microservice peut être développé à l'aide d'un langage et d'un cadre de programmation qui conviennent le mieux au problème que le microservice est conçu pour résoudre.
- Chaque microservice peut utiliser sa propre base de données.
- Chaque microservice est déployé indépendamment, sans affecter les autres microservices de l'application.
- Les microservices sont simples, ciblés et indépendants. L'application est donc plus facile à entretenir.
- Les fonctionnalités de l'application sont réparties entre plusieurs services. Si un microservice échoue, la fonctionnalité offerte par les autres microservices continue d'être disponible.

Pour une bonne gestion de ce projet, nous optons pour la méthode (ou processus) unifiée **2TUP**. **2TUP** (*2 Track Unified Process*) est un processus de développement logiciel qui implémente le Processus Unifié.

I.5 Cadre méthodologique : le processus unifié 2TUP

I.5.1 Processus Unifié

Le processus unifié est un processus de développement logiciel itératif, centré sur l'architecture, piloté par des cas d'utilisation et orienté vers la diminution des risques.

C'est un patron de processus pouvant être adapté à une large classe de systèmes logiciels, à différents domaines d'application, à différents types d'entreprises, à différents niveaux de compétences et à différentes tailles de l'entreprise. [1]

Un processus unifié se distingue par ses caractéristiques suivantes [2] :

- **Itératif** : le logiciel nécessite une compréhension progressive du problème à travers des raffinements successifs et permet de développer une solution effective de façon incrémentale par des itérations multiples.
- **Piloté par les risques** : les causes majeures d'échec d'un projet logiciel doivent être écartées en priorité.
- **Centré sur l'architecture** : le choix de l'architecture logicielle est effectué lors des premières phases de développement du logiciel. La conception des composants du système est basée sur ce choix.
- **Conduit par les cas d'utilisation** : le processus est orienté par les besoins utilisateurs représentés par des cas d'utilisation.

Les activités de modélisation reposent sur UML. Ce langage de modélisation couvre les aspects structurels et dynamiques de l'architecture et de la conception des logiciels. Il facilite une modélisation par composants en utilisant une approche orientée objet. [3]

Dans la communauté objet, il existe plusieurs processus unifiés en vogue comme eXtreme Programming (XP) et Rational Unified Process (RUP). Dans notre étude, on a choisi de travailler avec le processus 2TUP ; parce qu'il couvre des projets de toute taille et il a pu faire une large place dans le domaine de la technologie et les risques des projets.

I.5.2 Le processus 2TUP

Le **2TUP** propose un cycle de développement en Y, qui dissocie les aspects techniques des aspects fonctionnels. Il commence par une étude préliminaire qui consiste essentiellement à

identifier les acteurs qui vont interagir avec le système à construire, les messages qu'échangent les acteurs et le système, à produire le cahier des charges et à modéliser le contexte (le système est une boîte noire, les acteurs l'entourent et sont reliés à lui, sur l'axe qui lie un acteur au système on met les messages que les deux s'échangent avec le sens).

Le processus s'articule ensuite autour de trois phases essentielles :

- une branche technique ;
- une branche fonctionnelle ;
- une phase de réalisation.

La branche fonctionnelle capitalise la connaissance du métier de l'entreprise. Cette branche capture des besoins fonctionnels, ce qui produit un modèle focalisé sur le métier des utilisateurs finaux.

La branche technique capitalise un savoir-faire technique et/ou des contraintes techniques. Les techniques développées pour le système sont indépendantes des fonctions à réaliser.

La phase de réalisation consiste à réunir les deux branches, permettant de mener une conception applicative et enfin la livraison d'une solution adaptée aux besoins [4].

La **figure 4** suivante détaille les étapes de développement des trois branches du processus 2TUP**.**

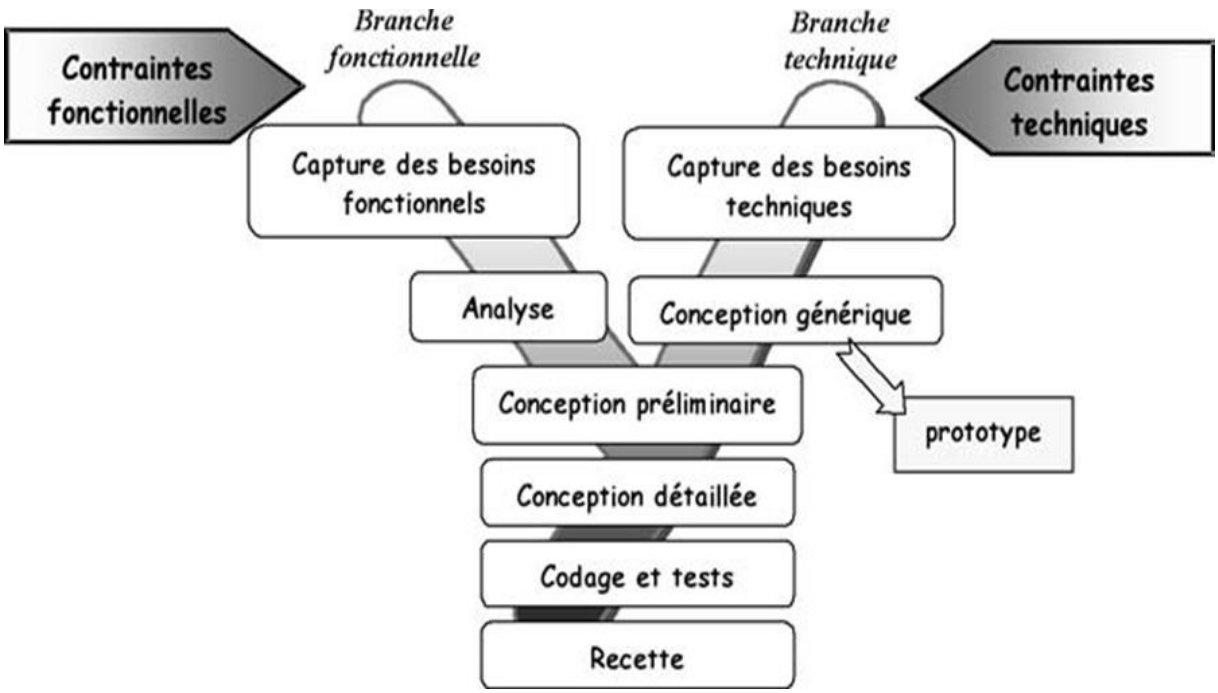


Figure 4 : Présentation du processus 2TUP

Conclusion

En résumé, ce chapitre nous a permis de définir le périmètre de notre sujet. Il a permis de faire une étude détaillée de l'existant, de fixer les objectifs de notre projet et de choisir la méthode à utiliser pour sa réalisation.

Dans le chapitre suivant, nous allons présenter la capture et l'analyse des besoins fonctionnels du système qui correspond aux deux premières étapes de la branche fonctionnelle du processus 2TUP.

CHAPITRE II : SPECIFICATION ET ANALYSE DES BESOINS FONCTIONNELS

La spécification et l'analyse des besoins représentent la première phase du cycle de développement d'un logiciel. Ainsi, dans ce chapitre, nous commençons d'abord par une spécification des besoins auxquels doit répondre l'application, ensuite à l'analyse des besoins à travers l'introduction des acteurs et les diagrammes de cas d'utilisation relatifs à ces acteurs.

II.1 Spécification des besoins fonctionnels

Cette phase consiste à comprendre le contexte du système. Il s'agit de déterminer les fonctionnalités et les acteurs les plus pertinents, de préciser les risques les plus critiques et d'identifier les cas d'utilisation initiaux.

II.1.1 Identification des acteurs

L'identification des acteurs est une étape importante dans un projet de développement. Elle permet de mieux mesurer l'influence de chacun des groupes d'acteurs sur l'action à mener. Le **tableau 1** ci-dessous présente les acteurs et leurs rôles.

Tableau 1 : Identification des acteurs et leurs taches correspondantes

Acteurs	**Rôles**
Etudiant	✓ S'inscrire ✓ Acheter des tickets de petit déjeuner et de repas ✓ Voir l'historique de ses achats et de ses dépenses ✓ Voir son nombre de ticket restant
Caissier ou Vendeur de ticket	✓ Vendre des tickets ✓ Voir les statistiques de ses ventes ✓ Ajouter un étudiant
Portier ou Contrôleur de l'accès au restaurant	✓ Débiter un ticket pour chaque étudiant qui accède au restaurant ✓ Voir les statistiques des accès au restaurant

Administrateur	✓ Ajouter les membres de l'administration qui devront utiliser la plateforme ✓ Attribuer les droits d'accès de chaque utilisateur ✓ Modifier, Supprimer ou désactiver les utilisateurs

II.1.2 Les diagrammes de cas d'utilisation

En langage UML, les diagrammes de cas d'utilisation modélisent le comportement d'un système et permettent de capturer ses exigences. Ils décrivent les fonctions générales et la portée d'un système. Ces diagrammes identifient également les interactions entre le système et ses acteurs. Les cas d'utilisation et les acteurs dans les diagrammes de cas d'utilisation décrivent ce que le système fait et comment les acteurs l'utilisent, mais ne montrent pas comment le système fonctionne en interne.

Pour ce travail, chaque acteur possède un diagramme de cas d'utilisation. Les **figures 5, 6, 7 et 7** suivantes représentent les diagrammes de cas d'utilisations des différents acteurs du système.

❖ **Diagramme de cas d'utilisation de la Gestion des achats de tickets**

L'acteur principal pour la gestion des achats des tickets est l'étudiant. Ce diagramme nous montre les différentes tâches que peut faire l'étudiant dans le système. L'étudiant peut acheter des tickets de restauration, consulter ses tickets restant et l'historique de ses achats et voire l'ensemble de ses dépenses en tickets. Mais pour se faire, il devra s'inscrire et s'authentifier par la suite pour accéder à ces fonctionnalités. La **figure 5** ci-dessous présente ce diagramme de cas d'utilisation.

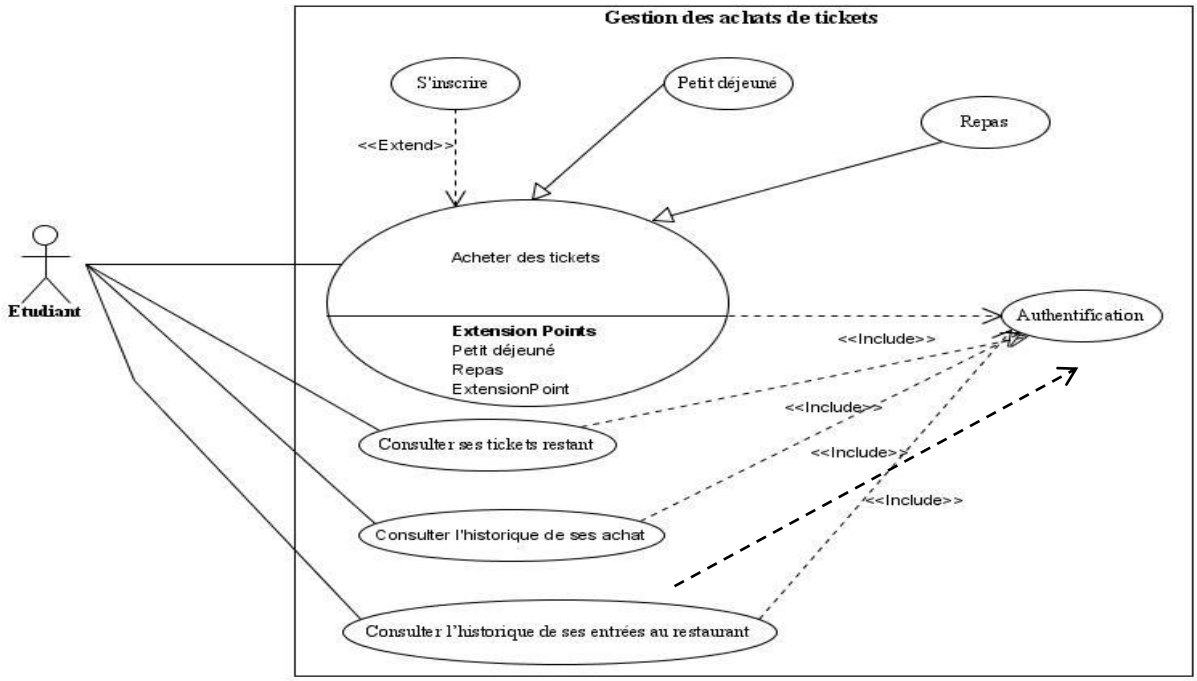


Figure 5 : Diagramme de cas d'utilisation de la Gestion des achats de tickets

❖ Diagramme de cas d'utilisation du Contrôle d'accès au restaurant

L'acteur principal pour le contrôle d'accès au restaurant est le contrôleur des accès au restaurant. Ce diagramme nous montre les différentes tâches que peuvent faire le contrôleur dans le système. Le contrôleur se charge de scanner le code QR qui identifie l'étudiant. Le fait de scanner ce code QR lui permet de débiter un ticket à l'étudiant. Il peut aussi voir le nombre d'accès au restaurant. Pour avoir accès à toutes ces fonctionnalités, il devra être inscrit par l'administrateur et s'authentifier. La **figure 6** ci-dessous présente ce diagramme de cas d'utilisation.

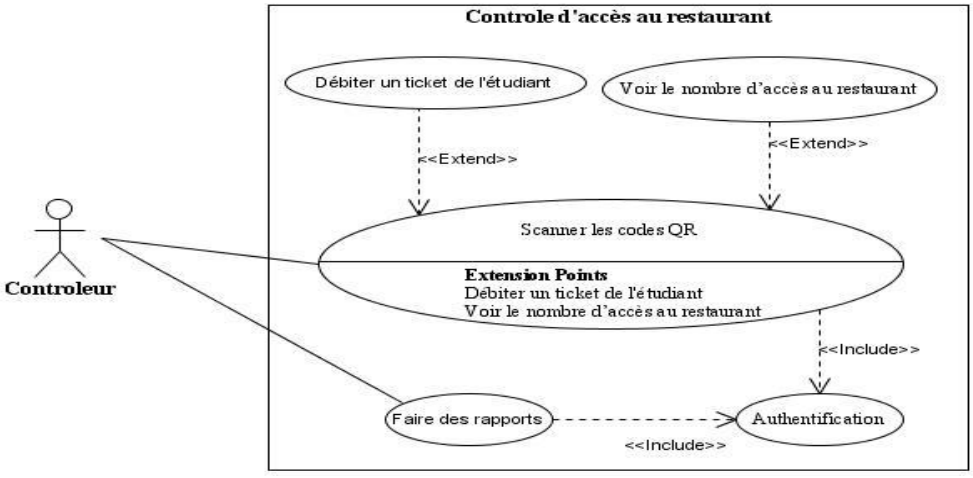


Figure 6 : Diagramme de cas d'utilisation du Contrôle d'accès au restaurant

❖ Diagramme de cas d'utilisation de la vente de ticket

L'acteur principal pour la vente de ticket est le caissier ou vendeur de ticket. Ce diagramme nous montre les différentes tâches que peut faire le caissier dans le système.

Le caissier peut vendre des tickets aux étudiant, suivre les ventes et leurs statistiques. Il peut aussi inscrire des étudiants. Pour avoir accès à toutes ces fonctionnalités, il doit être inscrit et s'authentifier. La **figure 7** ci-dessous présente ce diagramme de cas d'utilisation.

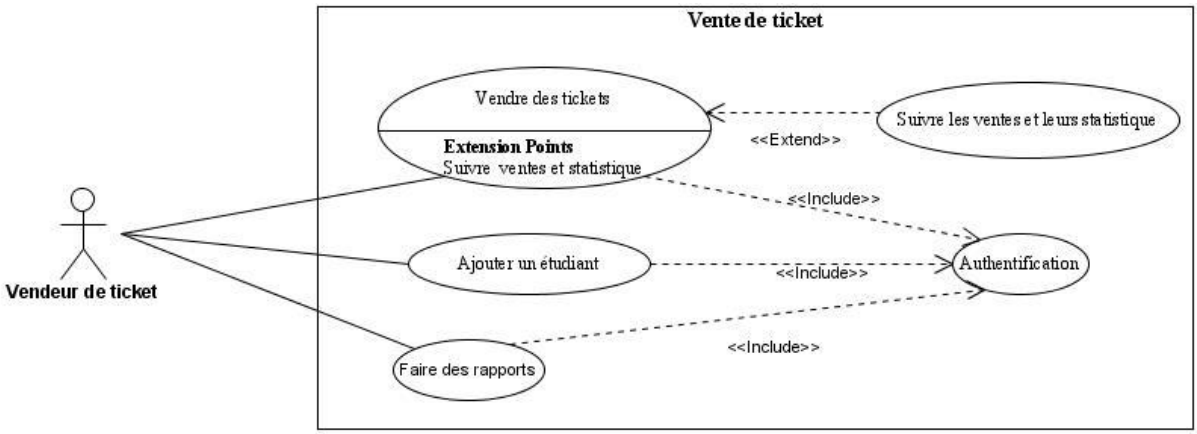


Figure 7 : Diagramme de cas d'utilisation de la vente de ticket

❖ Diagramme de cas d'utilisation Gestion des statistiques des accès au restaurant

Les principaux acteurs qui peuvent voir les statistiques des accès au restaurant sont les contrôleurs. Ce diagramme nous montre que les contrôleurs peuvent suivre le nombre d'étudiant qui ont accédé au restaurant et faire des rapports. Pour accéder à ces fonctionnalités ils devront se connecter d'abord. La **figure 8** ci-dessous présente ce diagramme de cas d'utilisation.

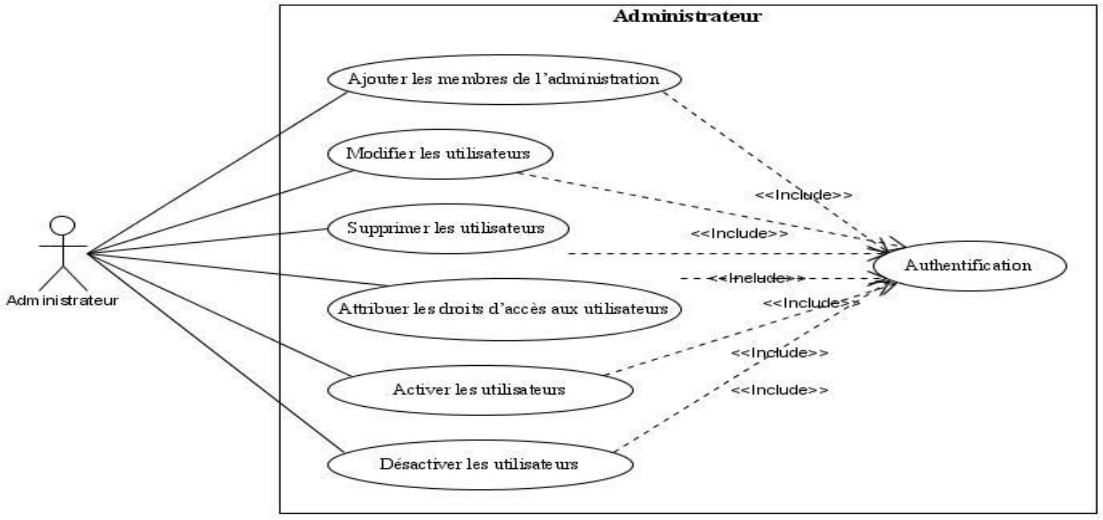


Figure 8 : Diagramme de cas d'utilisation Gestion des statistiques des accès au restaurant

II.2 Analyse des besoins fonctionnels

Dans le processus de développement d'un logiciel, l'analyse constitue une étape très importante. Elle ouvre le système pour établir la structure des objets d'analyse avec des diagrammes d'activité et de séquence.

Le diagramme de séquence est la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique dans la formulation UML.

Le diagramme d'activité est un diagramme comportemental d'UML, permettant de représenter le déclenchement d'événements en fonction des états du système.

Dans cette partie, nous allons faire l'analyse des besoins fonctionnels en décrivant les activités des fonctionnalités du système.

Plusieurs scénarii seront décrits. Il y'a deux scénarii possibles : un scénario normal et un scénario d'erreur. On parle de scénario normal lorsque tout se passe bien. Un scénario d'erreur est lorsqu'on rencontre une erreur lors de l'exécution d'une tache ou d'une fonctionnalité.

II.2.1 Achat de ticket

Pour la fonctionnalité **achat de ticket** plusieurs scénarii peuvent être observés : scénario normal et scénario d'erreur. La numérotation respecte le processus qui regroupe l'ensemble des taches consécutives effectués pour un achat de ticket.

Tableau 2 : Description du cas d'utilisation Achat de ticket

Nom	**Achat de ticket**
Résumé	Cette fonctionnalité permet l'achat de ticket de restaurant en ligne grâce aux solutions de paiement en ligne
Acteur	Etudiant
Précondition	Il faut être authentifier et être étudiant pour accéder à la fonctionnalité
Scénario normal	1. Le système affiche la page contenant le formulaire d'achat de ticket 2. L'étudiant rempli le formulaire en indiquant le nombre de ticket de repas et de petit déjeuner qu'il doit acheter

	3. Le système vérifie si le formulaire est bien rempli 4. Le système propose à l'étudiant les solutions de paiements en ligne disponible 5. L'étudiant choisi une solution 6. Le porte money électronique vérifie si l'étudiant a un solde suffisant pour effectuer l'achat 7. Le porte money électronique effectue la transaction et le système valide l'achat en ajoutant le nombre de ticket acheté au compte de l'étudiant
Exceptions	Il y'a exception lorsque le formulaire n'est pas bien rempli ou lorsque le solde de l'étudiant est insuffisant
Post condition	Achat validé avec un message d'information de l'achat effective

II.2.1.1 Diagramme d'activité

Le diagramme ci-dessous décrit l'ensemble des scénarii possible cité ci-dessus lors d'un achat de ticket.

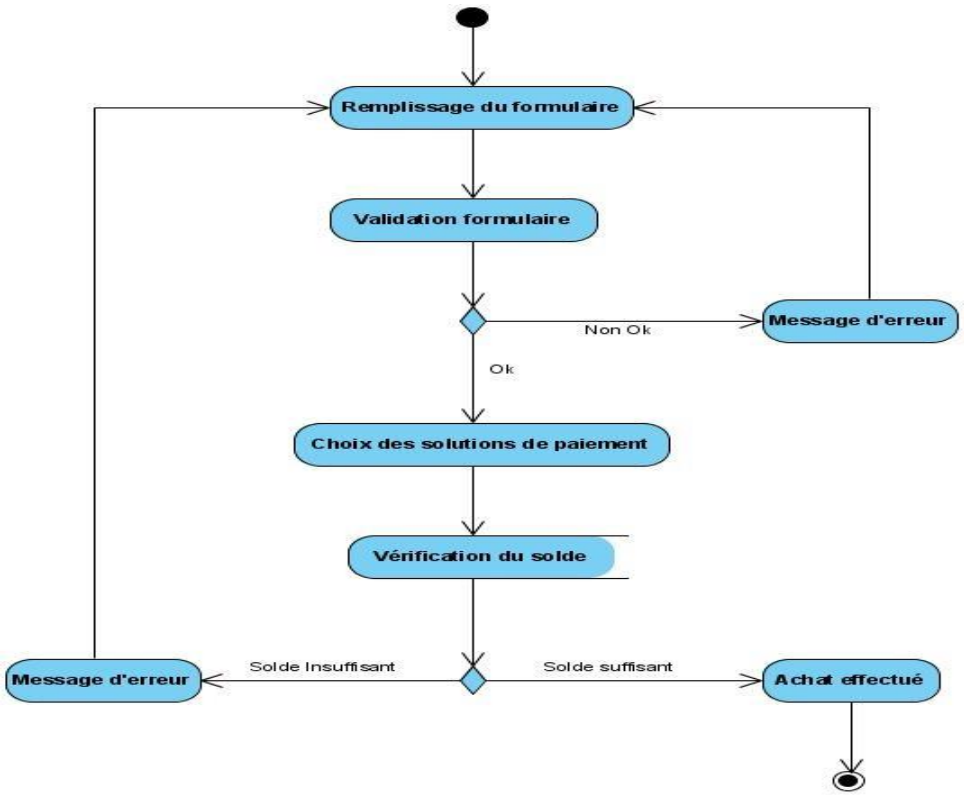


Figure 9 : Digramme d'activité achat de ticket

II.2.1.2 Diagramme de séquence

Le diagramme ci-dessous décrit le scénario normal cité ci-dessus pour l'achat de ticket.

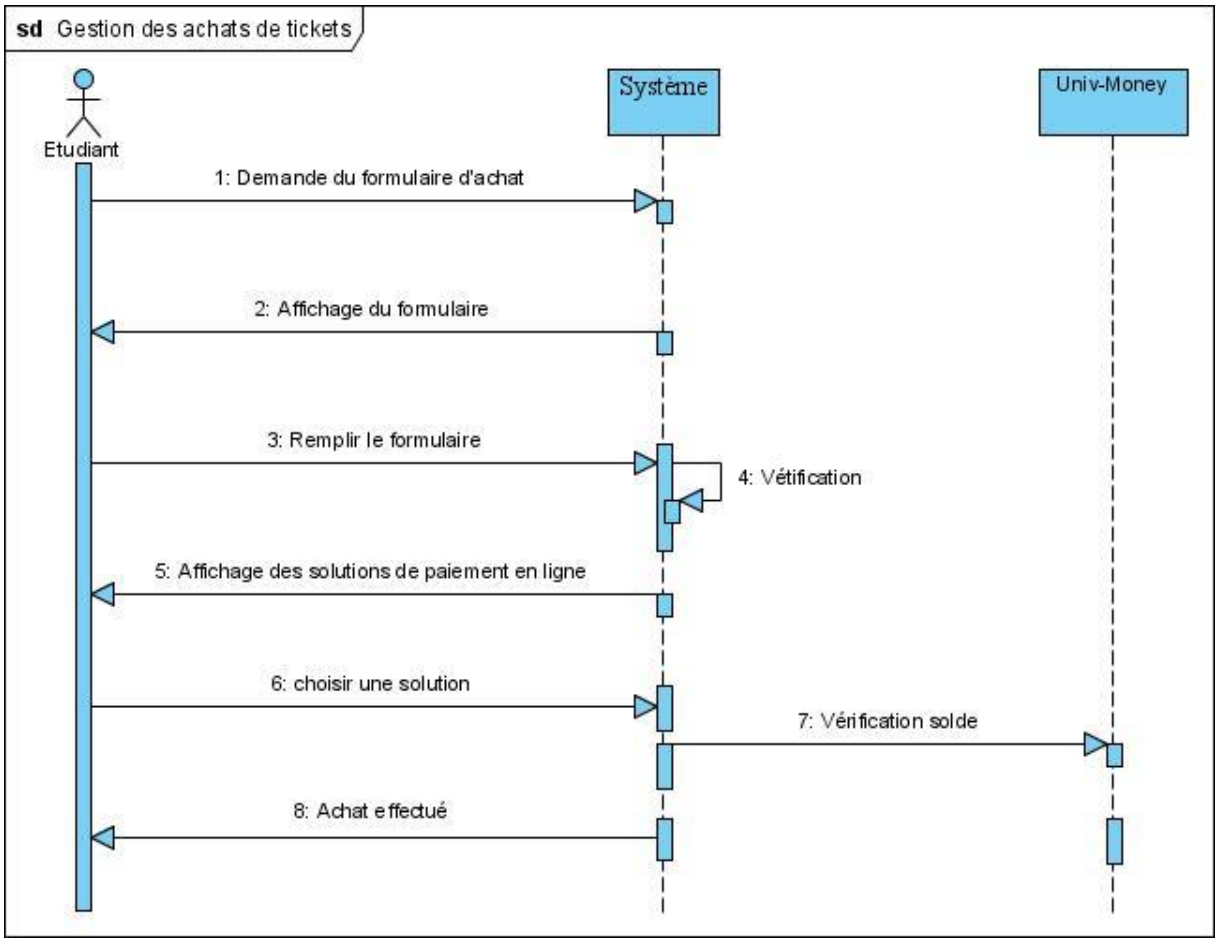


Figure 10 : Digramme de séquence Achat de ticket

II.2.2 Vente de ticket

Pour la fonctionnalité **vente de ticket** plusieurs scénarii peuvent être observer : scénario normal et scénario d'erreur. La numérotation respecte le processus qui regroupe l'ensemble des tache consécutifs effectué pour une vente de ticket.

Tableau 3 : Description du cas d'utilisation vente de ticket

Nom	**Vente de ticket**
Résumé	Cette fonctionnalité permet de vendre des tickets de restaurant
Acteur	Vendeur de ticket (caissier)
Précondition	Il faut être authentifier et être vendeur pour accéder à la fonctionnalité
Scénario normal	1. Le système affiche la page contenant le formulaire d'achat de ticket 2. Le caissier (vendeur de ticket) renseigne le nombre de repas et de petit déjeuner que l'étudiant souhaite acheter et le numéro INE de l'étudiant 3. Le vendeur de ticket appuie sur le bouton « Valider » 4. Le système vérifie la conformité des informations saisies dans chaque champ 5. Le système vérifie si l'étudiant existe 6. Le système valide et effectue l'achat
Exceptions	Il y'a exception lorsque qu'il y a des erreurs de saisie ou lorsque l'étudiant n'existe pas
Post condition	Vente effectuée avec un message d'information de la vente effective

II.2.2.1 Diagramme d'activité

Le diagramme ci-dessous décrit l'ensemble des scénarii possible cité ci-dessus lors d'une vente de ticket.

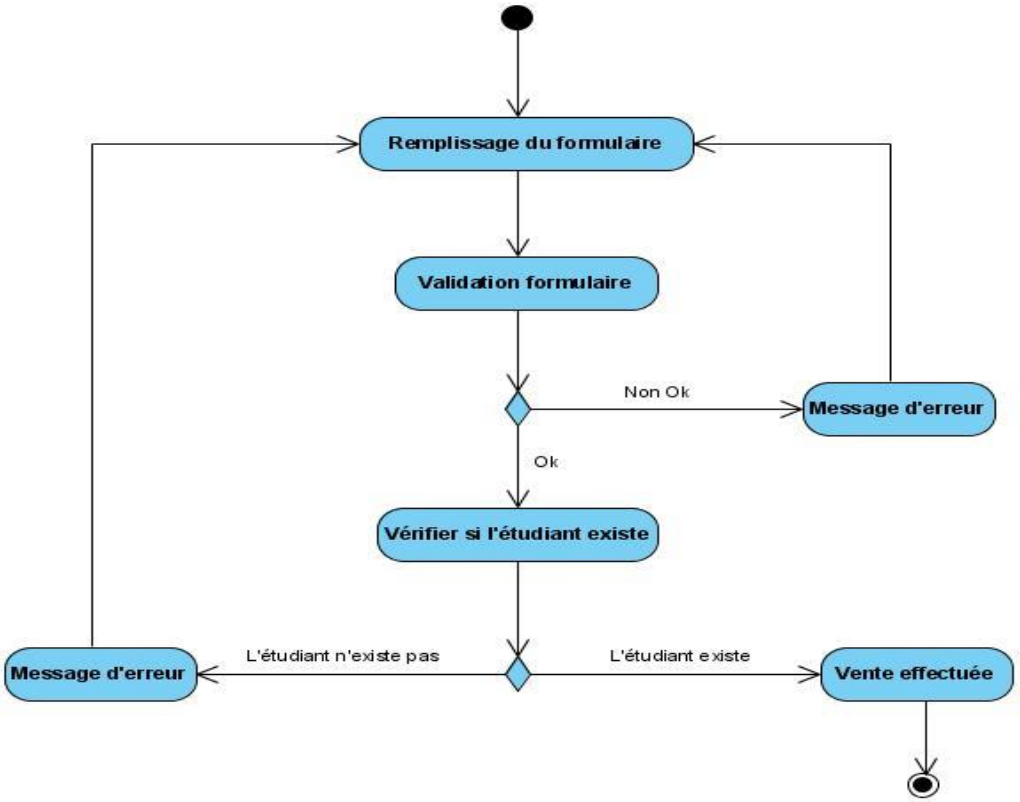


Figure 11 : Diagramme d'activité vente de ticket

II.2.2.2 Diagramme de séquence

Le diagramme ci-dessous décrit le scénario normal cité ci-dessus pour la vente de ticket.

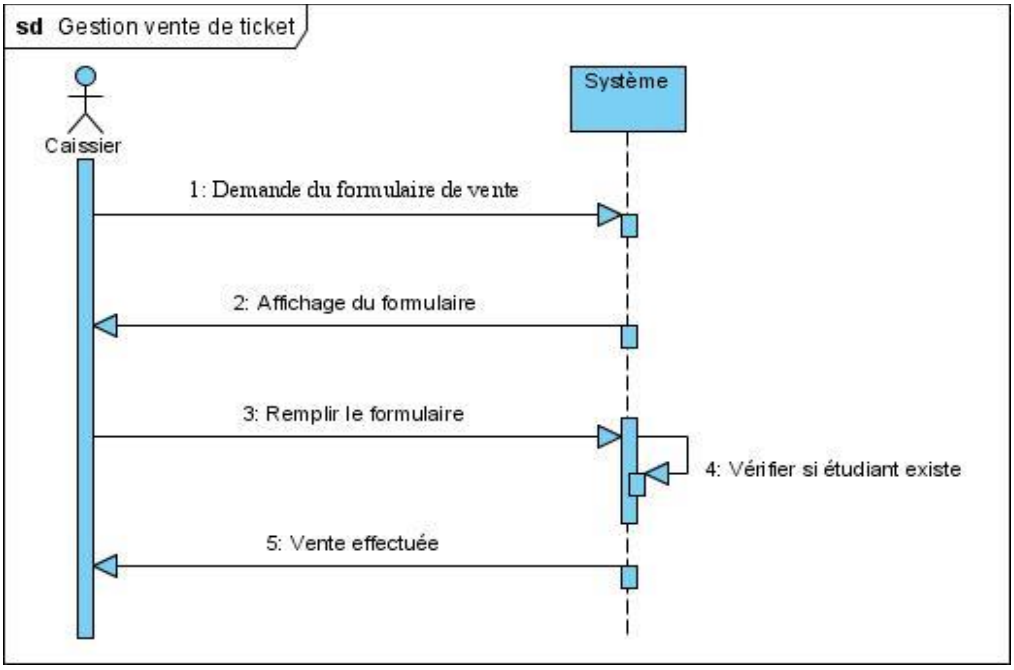


Figure 12 : Diagramme de séquence vente de ticket

II.2.3 Contrôle des accès au restaurant

Pour la fonctionnalité **contrôle des accès au restaurant** plusieurs scénarii peuvent être observer : scénario normal et scénario d'erreur. La numérotation respecte le processus qui regroupe l'ensemble des tache consécutifs effectué pour le contrôle d'accès au restaurant.

Tableau 4 : Description du cas d'utilisation contrôle des accès au restaurant

Nom	**Contrôle des accès au restaurant**
Résumé	Cette fonctionnalité permet de vérifier si l'étudiant rempli les conditions pour accéder au restaurant
Acteur	Contrôleur (portier)
Précondition	Il faut être authentifier et être un controleur pour accéder à la fonctionnalité
Scénario normal	1. Le portier accède à l'application 2. L'application lui permet de scanner le code QR de l'étudiant 3. Après scanne, le système vérifie si le code QR est valide 4. Le système vérifie si l'étudiant a au moins un ticket qui correspond au repas
Exceptions	Il y'a exception lorsque le code QR que l'étudiant à présenter ne correspond à aucun étudiant ou lorsque l'étudiant à épuiser ses tickets
Post condition	Le système débite un ticket à l'étudiant et ajoute +1 au nombre d'entré au restaurant

II.2.3.1 Diagramme d'activité

Le diagramme ci-dessous décrit l'ensemble des scénarii possible cité ci-dessus lors des contrôles d'accès au restaurant.

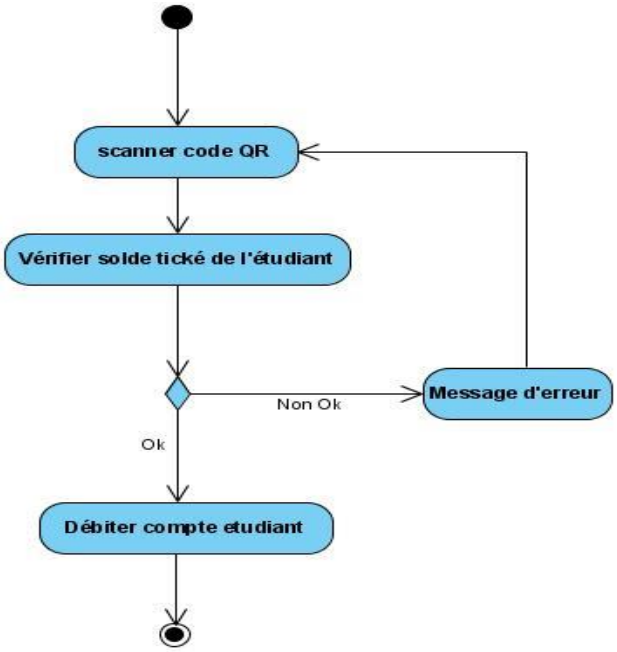


Figure 13 : Diagramme d'activité contrôle accès au restaurant

II.2.3.2 Diagramme de séquence

Le diagramme ci-dessous décrit le scénario normal cité ci-dessus pour le contrôle d'accès au restaurant.

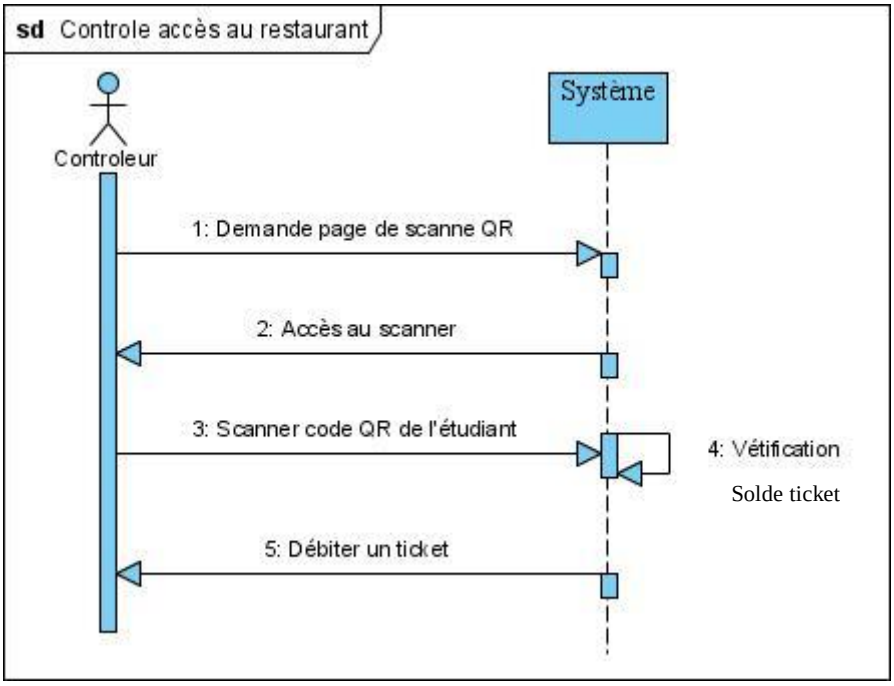


Figure 14 : Diagramme de séquence contrôle accès au restaurant

Conclusion

Ce chapitre nous a permis de faire la spécification et l'analyse des besoins fonctionnels nécessaire à la conception du projet. Ainsi nous passons au chapitre suivant qui consiste à la conception du système en suivant toujours le processus 2TUP.

CHAPITRE III : CONCEPTION DU SYSTEME

Dans ce chapitre, nous allons aborder d'abord la conception générale dans laquelle nous parlerons de l'architecture logicielle du système et des différents diagrammes, dont celui de composant, de package et de déploiement et enfin parler de la conception détaillée qui résulte de la mise en place du diagramme de classe et de l'élaboration du dictionnaire des données.

III.1 Conception générale

III.1.1 Architecture logicielle du système

L'architecture logicielle décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions.

Pour ce projet, nous allons utiliser une architecture microservice. Les architectures de microservices sont la « nouvelle norme ». La création de petites applications autonomes et prêtes à l'emploi peut apporter une grande flexibilité et une résilience accrue à notre code. Les nombreuses fonctionnalités spécialement conçues pour Spring Boot facilitent la création et l'exécution des microservices en production à grande échelle. [5]

III.1.1.1 Découpage en microservices de notre système

Le principe est de mettre en place un système divisé en quatre (4) parties. Chaque partie représente un microservice.

1. Une partie pour les vendeurs de tickets.
2. Une partie pour permettre aux responsables du restaurant d'avoir les statistiques des accès au restaurant.
3. Une partie pour les étudiants pour la gestion de ses achats de tickets et de ses accès au restaurant.
4. Une partie pour les contrôleurs afin de contrôler les accès au restaurant.

Détails :

1. **La partie pour les vendeurs de tickets**

   Il sera constitué de :

   - Page de vente des tickets
   - Page de suivi et statistique des ventes (par jour, semaine, moi)
   - Une page pour les rapports journalier, hebdomadaire, et mensuel.
   - Page d'ajout d'un nouvel étudiant

2. **La partie pour permettre aux responsables du restaurant d'avoir les statistiques des accès au restaurant**

   Il sera constitué de :

   - Une page pour les permettre de suivre en temps réel les statistiques des accès au restaurant.
   - Une page pour les rapports journalier, hebdomadaire, et mensuel.

3. **La partie pour les étudiants pour la gestion de ses achats de tickets et de ses accès au restaurant**

   Il sera constitué de :

   - Un code QR qui permet d'identifier chaque étudiant
   - Une page pour acheter des tickets (Orange Money, Univ-Money, ...)
   - Une page pour consulter l'historique de ses achats et de ses accès au restaurant.
   - Une page pour consulter ses tickets restant (petit déjeuné et repas)

4. **La partie pour les contrôleurs afin de contrôler les accès au restaurant**

   Il sera constitué de :

   - Un scanner de code QR pour les contrôles d'accès au restaurant
   - Une page pour voir le nombre d'accès au restaurant par repas et par jour.
   - Une page pour les rapports journalier, hebdomadaire, et mensuel.

III.1.1.2 Architecture de notre système

Les Edge Microservices permettant à nos Microservices d'être complètement adapté au cloud. Il s'agit des microservices **1**, **2** et **3** ci-dessous. Ils vont résoudre des problèmes comme :

- Comment Trouver les instances d'un Microservices ?
- Comment Equilibrer la répartition de la charge ?
- Comment Configurer et sécuriser l'application ?
- Comment Débugger et surveiller l'application ?

Microservice 1 : *Config Server*

C'est un service de configuration, dont le rôle est de centraliser les fichiers de configuration des différents microservices dans un endroit unique.

Chaque Microservice que nous créons à un certain nombre de configuration. L'ensemble des fichiers de configurations pour cette architecture sont centralisés dans GitHub.

Considérons qu'après déploiement de nos Microservices, qu'on veut changer un ou des paramètres de configuration d'un Microservices qui a plusieurs instances. On sera obligé d'arrêter toutes les instances, ce qui n'est pas une bonne pratique. La solution est d'externaliser tous ces fichiers de configuration dans un serveur central. La **figure 15** ci-dessous permet d'illustrer.

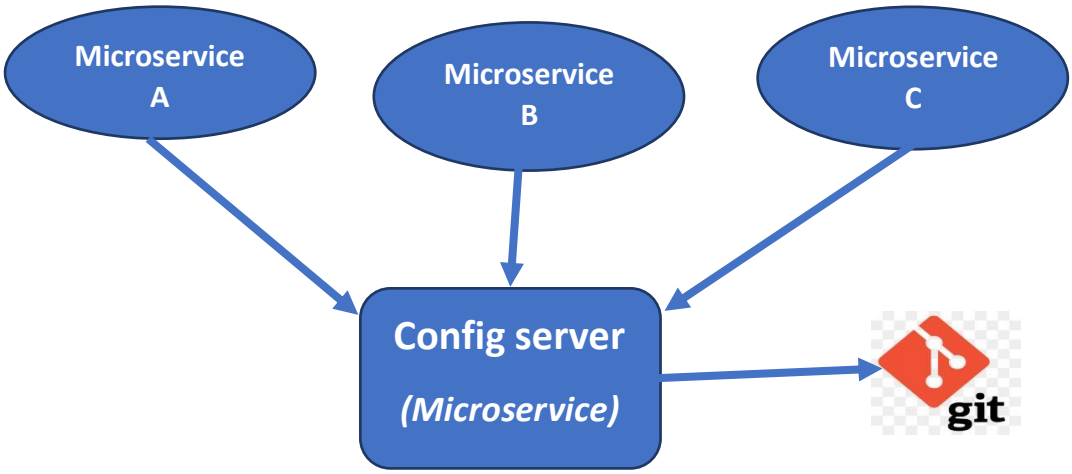


Figure 15 : Interaction entre le microservice config-server et les autres microservices

Microservice *2 : Discovery Server*

C'est un service permettant l'enregistrement des instances de microservices en vue d'être découvertes par d'autres microservices.

Quand un microservice doit être très sollicité, il faut en démarrer plusieurs instances pour répondre à la forte demande. Ce service va permettent l'enregistrement des instances de microservices en vue d'être découvertes par d'autres Microservices.
Pour éviter un couplage fort entre microservices, il est fortement recommandé d'utiliser un service de découverte qui permet d'enregistrer les propriétés des différents services et d'éviter ainsi d'avoir à appeler un service directement. Au lieu de cela, le service de découverte fournira dynamiquement les informations nécessaires, ce qui permet d'assurer l'élasticité et la dynamicité propres à notre architecture microservices. La **figure 16** ci-dessous nous permet de l'illustrer.

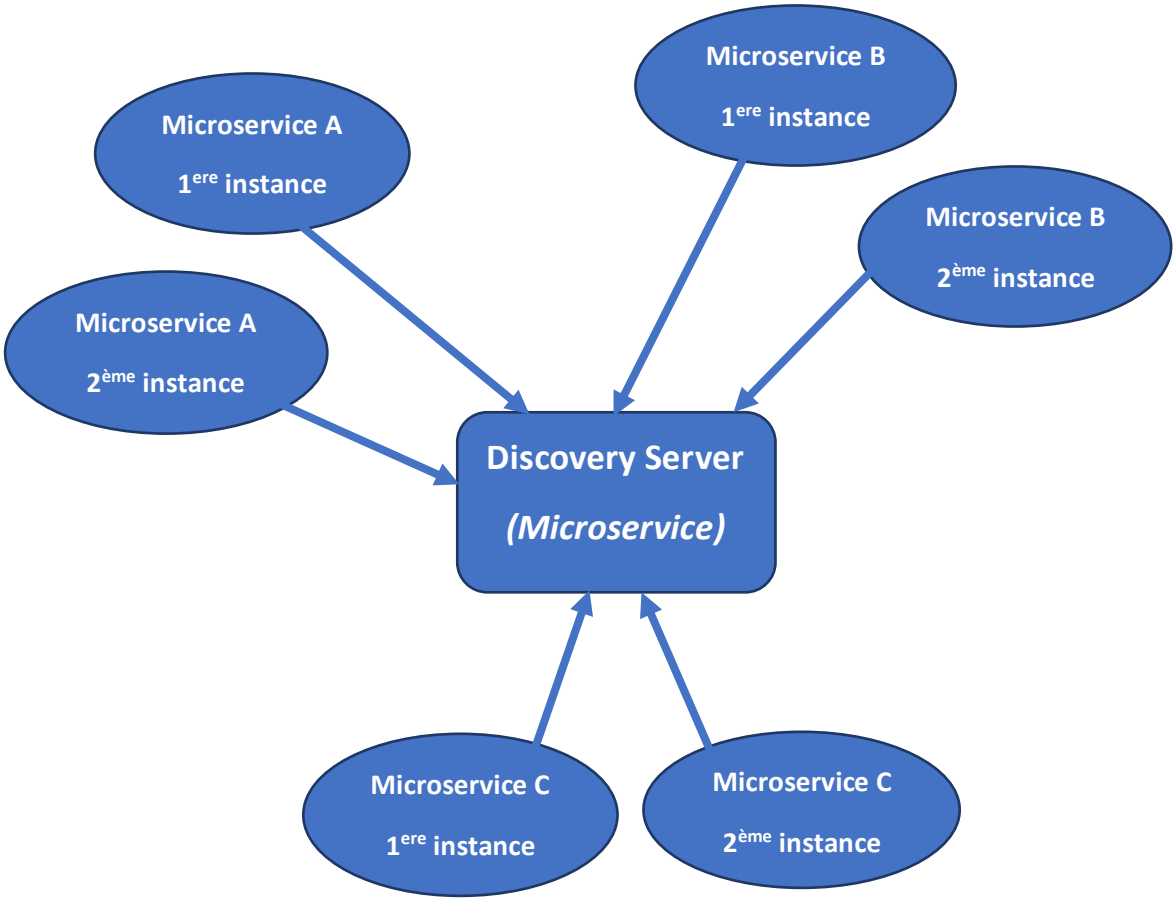


Figure 16 : Interaction entre le microservice discovery-server et les autres microservices

Microservice *3 : Proxy Server*

C'est une passerelle qui se charge du routage d'une requête vers l'une des instances d'un microservice, de manière à gérer automatiquement la distribution de charge.
Si on prend l'exemple d'une application comportant plusieurs Microservices, Ce qui est notre cas, l'implémentation de la sécurisation de tous les Microservices peut être problématique quand on veut dans l'avenir changer le protocole de sécurité de l'application. Un point d'entrée unique pourrait régler le problème. La **figure 17** ci-dessous permet de l'illustrer.

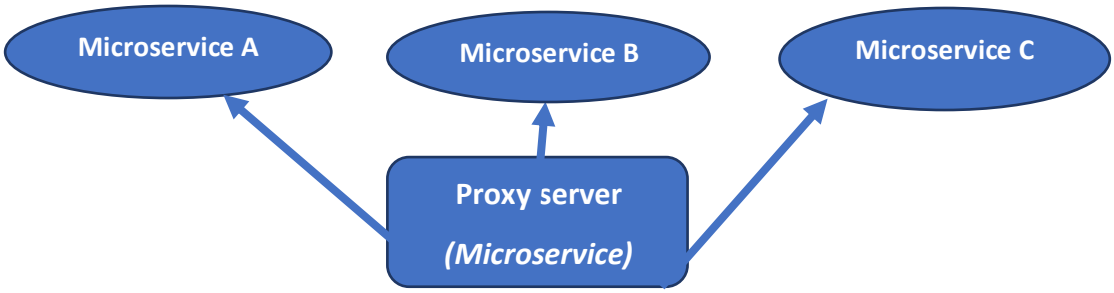


Figure 17 : Interaction entre le microservice proxy-server et les autres microservices

Microservice 4 : vendeur de tickets

C'est le microservice qui implémente la partie pour les vendeurs de tickets.

Microservice 5 : statistiques des accès au restaurant

C'est le microservice qui implémente la partie qui permet aux responsables du restaurant de suivre les statistiques des accès au restaurant.

Microservice 6 : étudiant

C'est le microservice qui implémente la partie permettant aux étudiants de gérer la gestion de leurs accès au restaurant et de leurs achats de tickets via la porte money **Univ-Money** qui a été développé par Monsieur Mor MBOUP étudiant à l'Université Assane Seck de Ziguinchor lors de son mémoire intitulé « Univ-money, un porte-monnaie électronique pour la gestion des transactions étudiantes au sein de l'université ».

Microservice 7 : contrôle d'accès au restaurant

C'est le microservice qui implémente la partie qui permet aux contrôleurs de contrôler les accès au restaurant.

L'architecture résultante :

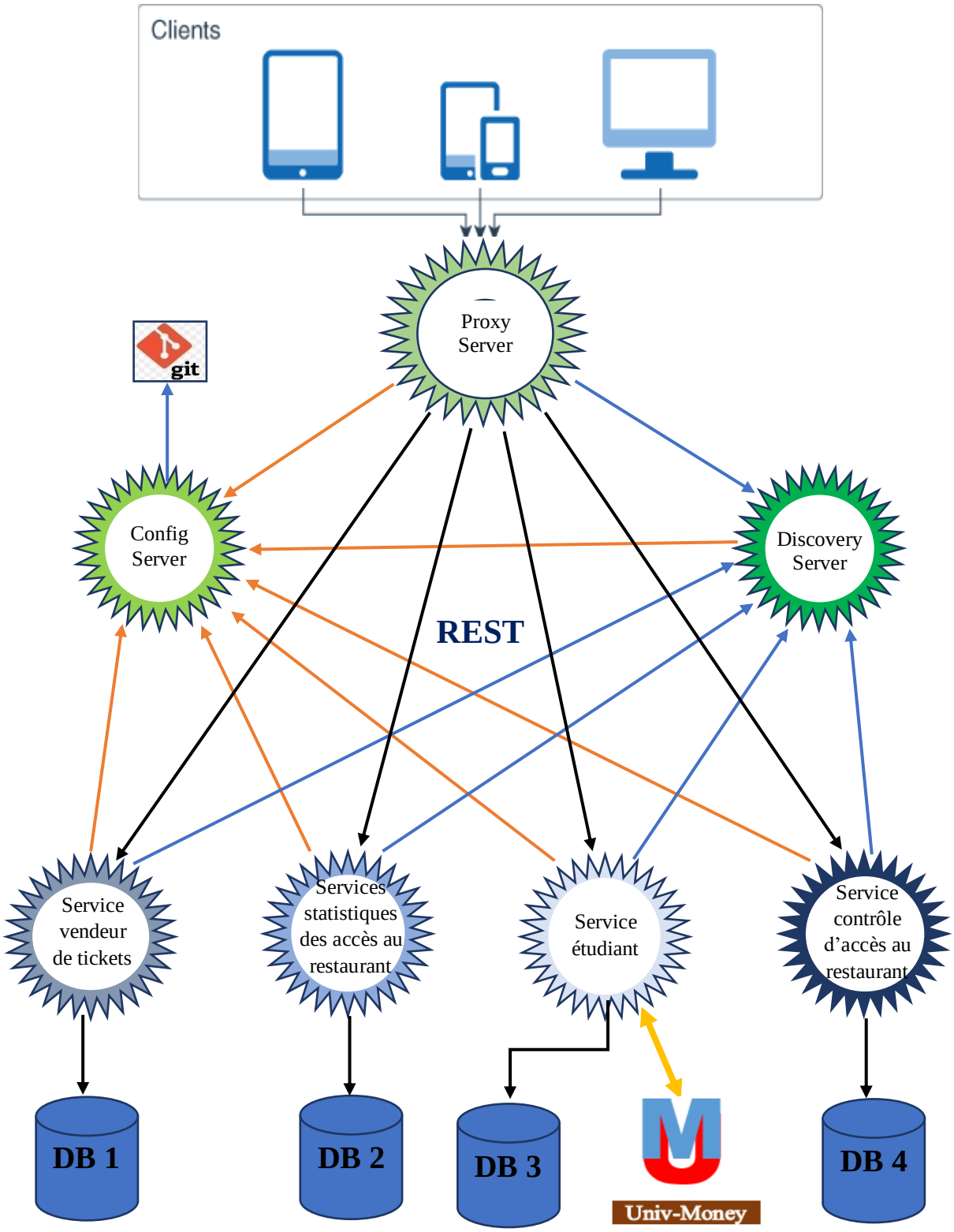


Figure 18 : Architecture globale du système

III.1.2 Diagramme de composants

Un diagramme de composants a pour objectif d'illustrer la relation entre les différents composants d'un système. Dans le cadre de l'UML 2.0, le terme « composant » fait référence à un module de classes qui représentent des systèmes ou des sous-systèmes indépendants ayant la capacité de s'interfacer avec le reste du système. [7]

Un étudiant n'ayant pas accès à son compte peut passer chez le vendeur, pour que ce dernier lui crédite son compte.

La **figure 19** suivante représente celle proposée pour notre système :

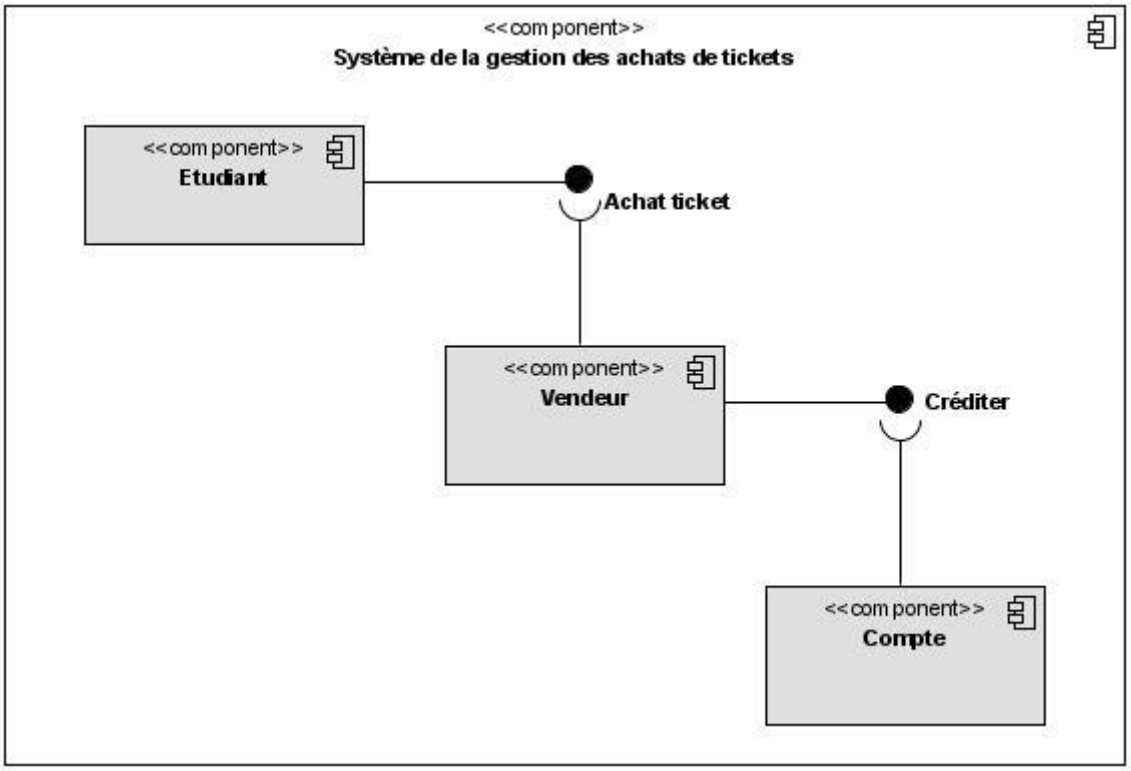


Figure 19 : Diagramme de composants Système de la gestion des achats de tickets

III.1.3 Diagramme de paquetage

Les diagrammes de package (ou diagramme de paquetages) sont des diagrammes structurels utilisés pour représenter l'organisation et la disposition de divers éléments modélisés sous forme de paquetages. Un paquetage est un regroupement d'éléments UML apparentés, tels que des diagrammes, des documents, des classes ou même d'autres paquetages. Tous les éléments du diagramme sont imbriqués dans des paquetages, qui sont eux-mêmes représentés sous

forme de dossiers de fichiers et organisés de manière hiérarchique. Les diagrammes de paquetages sont le plus souvent utilisés pour donner un aperçu visuel de l'architecture en couches d'un classifieur UML, tel qu'un système logiciel. [8]

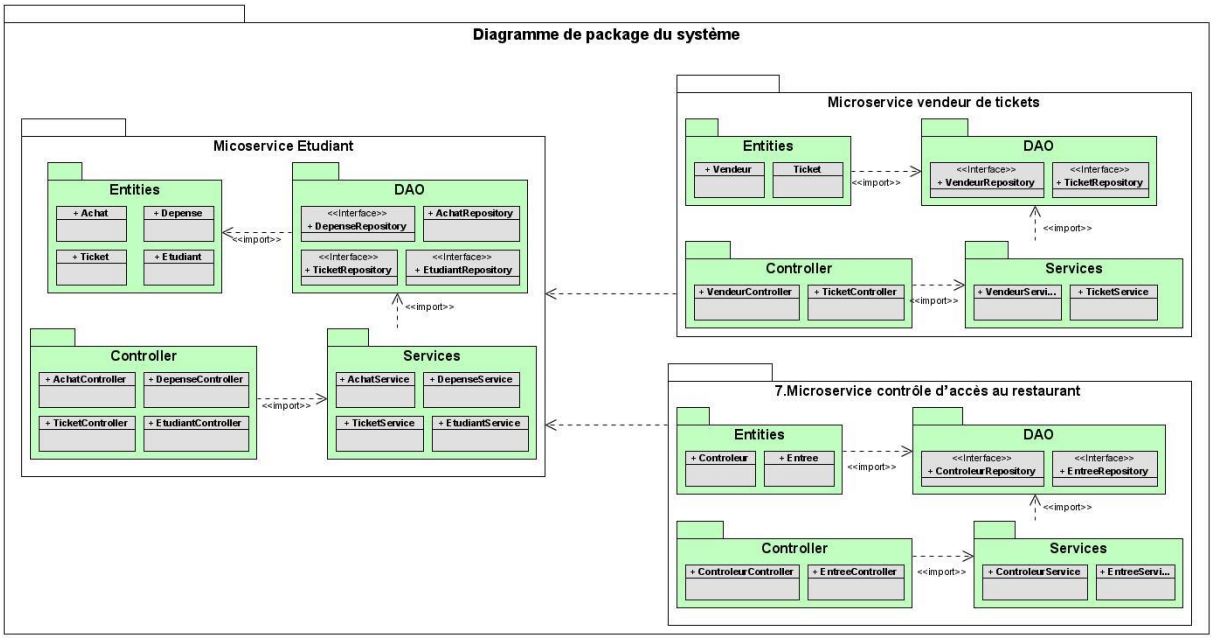


Figure 20 : Diagramme de package du système

III.1.4 Diagramme de déploiement

Le diagramme de déploiement UML montre l'architecture d'exécution de systèmes qui représentent l'affectation (déploiement) des artefacts logiciels à des cibles de déploiement. Il est utilisé pour visualiser la topologie des composants physiques d'un système dans lequel les composants logiciels sont déployés. Les diagrammes de déploiement sont très utiles pour décrire les composants matériels où les composants logiciels sont déployés. Le diagramme de déploiement permet également de modéliser l'aspect physique d'un système du logiciel orienté objet. [9]

Nous allons déployer chaque Microservice sur une machine séparée (voir figure 9). Les plateformes MySQL et Docker sont déployées sur des machines i5 avec au moins 4 GB de RAM qui évoluent sous le système d'exploitation Linux.

Docker Engine est une technologie de conteneurisation open source pour créer et conteneuriser des applications. Docker Engine agit comme une application client-serveur. [6]

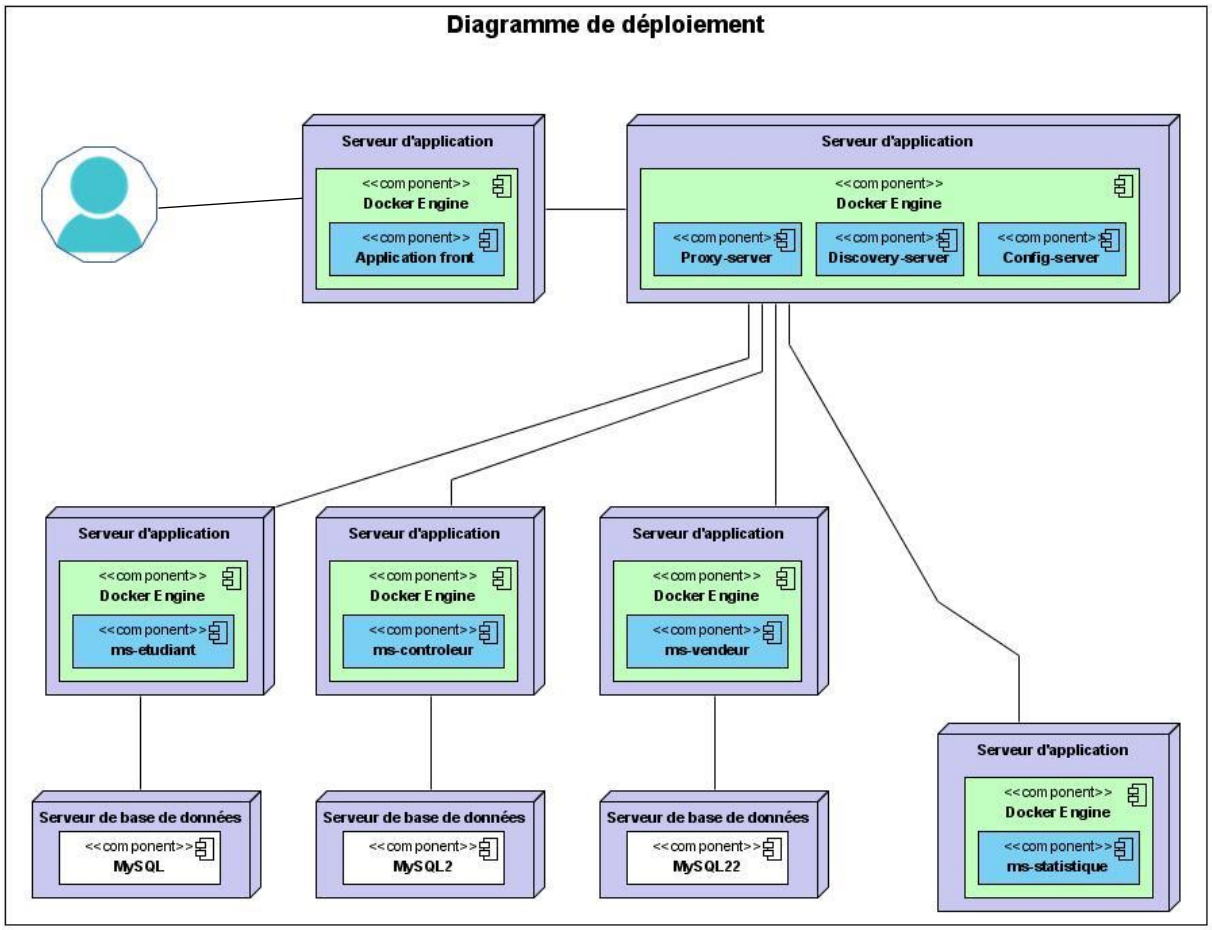


Figure 21 : Diagramme de déploiement

III.2 Conception détaillée

C'est le raffinement des éléments précédents jusqu'à l'obtention d'une forme permettant d'écrire immédiatement les programmes.

III.2.1 Dictionnaire de données

Le dictionnaire de données est le résultat de la phase de collecte des données. Cette phase est également appelée recueil d'information. En effet, pendant la phase de conception, les données recueillies et spécifiées sont inscrites dans un dictionnaire. Ce dictionnaire est un outil important, car il constitue la référence de toutes les études effectuées.

Le dictionnaire de donnée est un tableau qui regroupe toutes les données du SI, pour chaque donnée identifiée il faut préciser :

- Nom symbolique
- Signification ou description
- Type de retour

Le **tableau 5** ci-dessous représente l'ensemble des données utilisées pour élaborer ce système:

Tableau 5 : Dictionnaire de données

Nom des Classes	Nom symbolique	Signification ou description	Type de retour
Etudiant	Id	Identifiant de l'étudiant	Long
	numeroINE	Numéro d'Identifiant National Etudiant	String
	numeroEtud	Numéro de l'étudiant	String
	prenom	Prénom de l'étudiant	String
	nom	Nom de l'étudiant	String
	sexe	Sexe de l'étudiant	String
	dateNaissance	Date de naissance de l'étudiant	Date
	niveau	Niveau de l'étudiant	String
	filiere	Filière de l'étudiant	String
	telephone	Le numéro de téléphone mobile de l'étudiant	String
	email	L'adresse mail de l'étudiant	String
Ticket	id	Identifiant du ticket	Long
	petitDej	Nombre de petit déjeuner	Integer
	repas	Nombre de repas	Integer
Vendeur	Id	Identifiant du vendeur de ticket	Long
	prenom	Prénom du vendeur de ticket	String
	nom	Nom du vendeur de ticket	String
	sexe	Sexe du vendeur de ticket	String
	statut	Statut du vendeur de ticket	String
	telephone	Numéro de téléphone du vendeur de ticket	String
	email	Adresse mail du vendeur de ticket	String
	Id	Identifiant du contrôleur d'accès au RU	Long
	prenom	Prénom du contrôleur	String
	nom	Nom du contrôleur	String

Controleur	sexe	Sexe du contrôleur	String
	statut	Statut du contrôleur	String
	telephone	Numéro de téléphone du contrôleur	String
	email	Adresse mail du contrôleur	String
Achat	Id	Identifiant de l'achat	Long
	petiDej	Nombre de petit déjeuner acheté par un étudiant	Integer
	repas	Nombre de repas acheté par un étudiant	Integer
	dateAchat	Date de l'achat	Date
Depense	id	Identifiant de la dépense de ticket de l'étudiant	Long
	type	Entrée repas ou petit déjeuner	String
	nombre	Nombre de ticket dépensé par un étudiant	Integer
	dateDepense	Date de la dépense	Date

III.2.2 Diagramme de classes

Les diagrammes de classes sont l'un des types de diagrammes UML les plus utiles, car ils décrivent clairement la structure d'un système particulier en modélisant ses classes, ses attributs, ses opérations et les relations entre ses objets. Ainsi, les diagrammes de classes suivants représentent respectivement notre modèle opté pour ce projet.

III.2.2.1 Diagramme de classes participantes du système complet

Le diagramme de classes de la **figure 22** ci-dessous décrit l'ensemble des classes participantes du système complet sans prendre en compte le découpage en microservice.

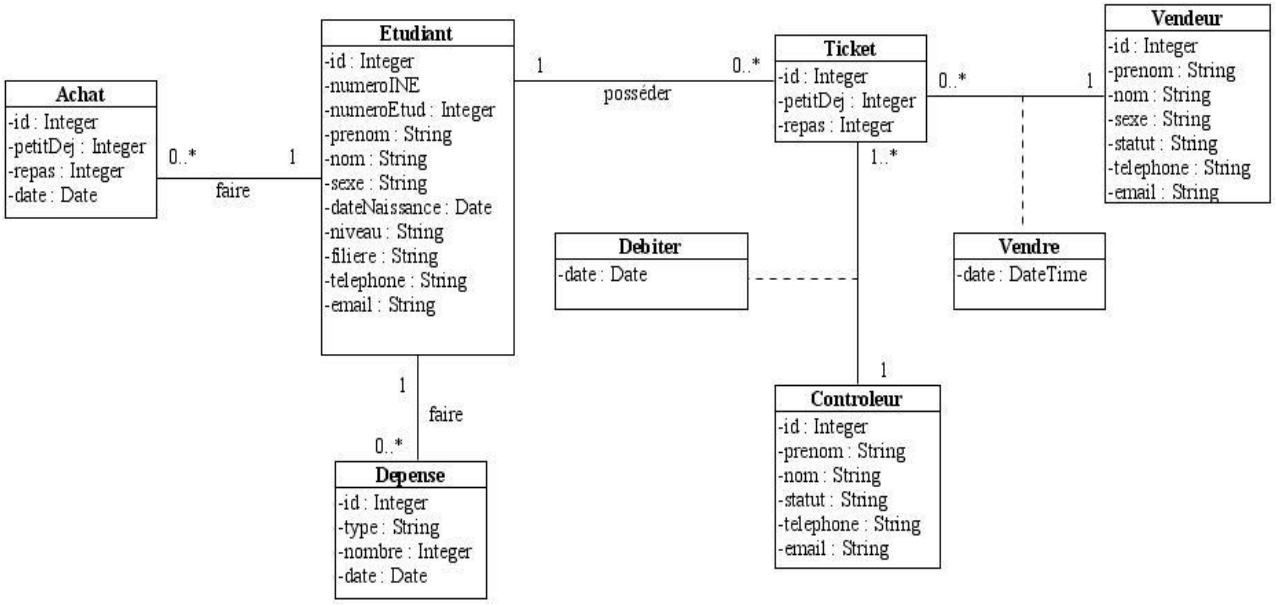


Figure 22 : Diagramme de classes du système complet

Ce diagramme montre qu'un étudiant peut posséder plusieurs tickets et peut faire plusieurs achat et dépense de ticket. Le vendeur de ticket peut faire plusieurs ventes et le contrôleur ou portier peut débiter plusieurs tickets.

La **figure 22** ci-dessus représente le diagramme de classe du système si on devait développer une application monolithique. Puisque notre architecture repose sur les microservice, nous devons faire un découpage logique et simple de ce diagramme. Chaque microservice doit avoir sa propre base de données donc auront chacun un digramme de classe correspondant.

III.2.2.2 Diagramme de classes pour le microservice étudiant

Le diagramme de classes de la **figure 24** ci-dessous décrit l'ensemble des classes participantes aux fonctionnalités du microservice étudiant.

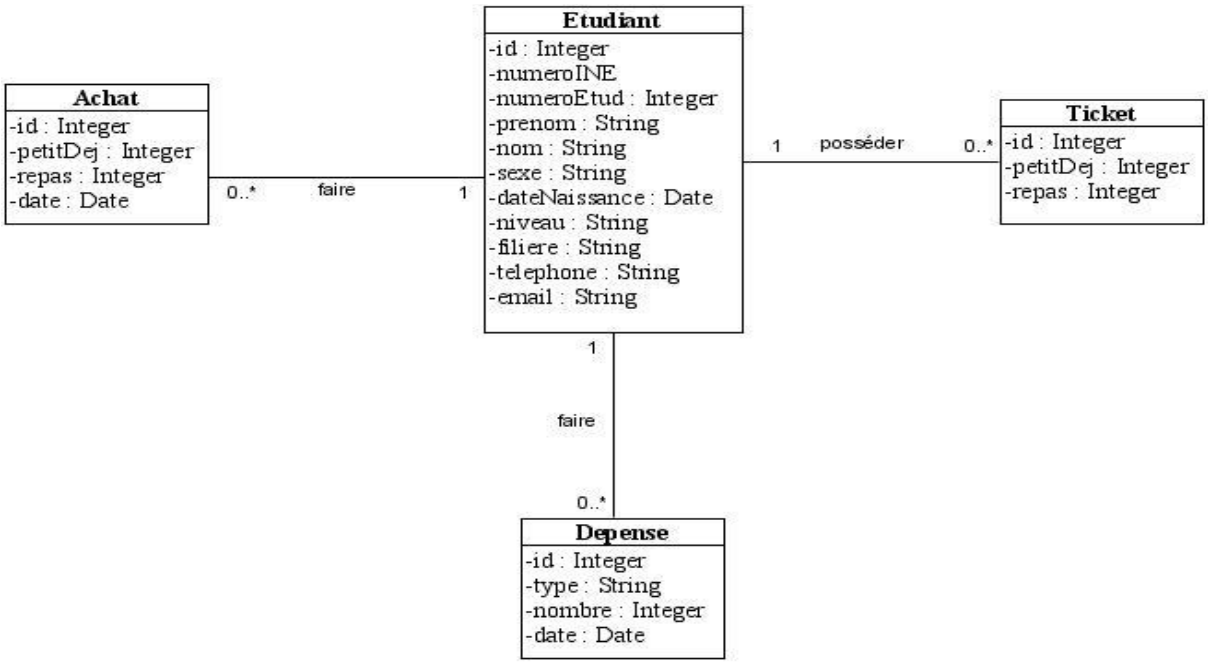


Figure 23 : Diagramme de classes pour le microservice étudiant

III.2.2.3 Diagramme de classes pour le microservice vendeur de tickets

Le diagramme de classes de la **figure 24** ci-dessous décrit l'ensemble des classes participantes aux fonctionnalités du microservice vendeur de tickets.

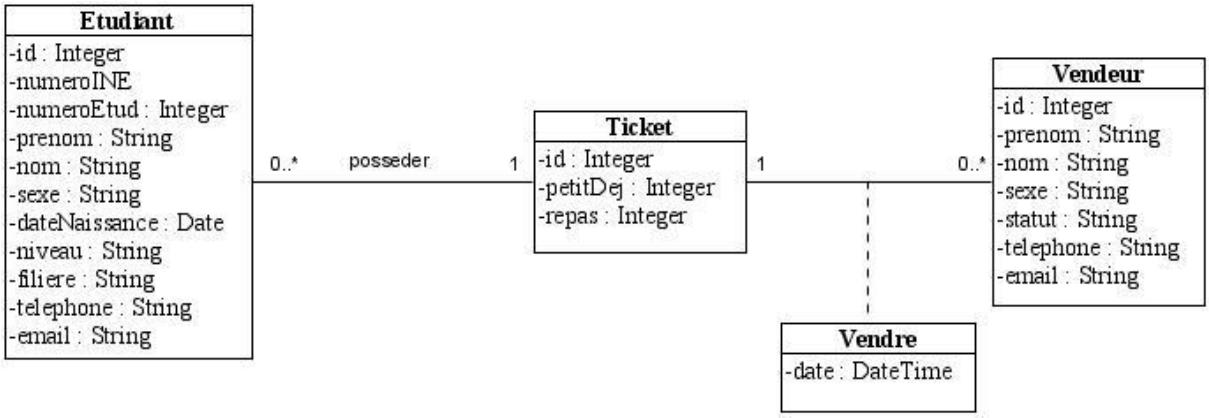


Figure 24 : Diagramme de classes pour le microservice vendeur de tickets

III.2.1.4 Diagramme de classes pour le microservice contrôle d'accès au restaurant

Le diagramme de classes de la **figure 25** ci-dessous décrit l'ensemble des classes participantes aux fonctionnalités du microservice Contrôle d'accès au restaurant.

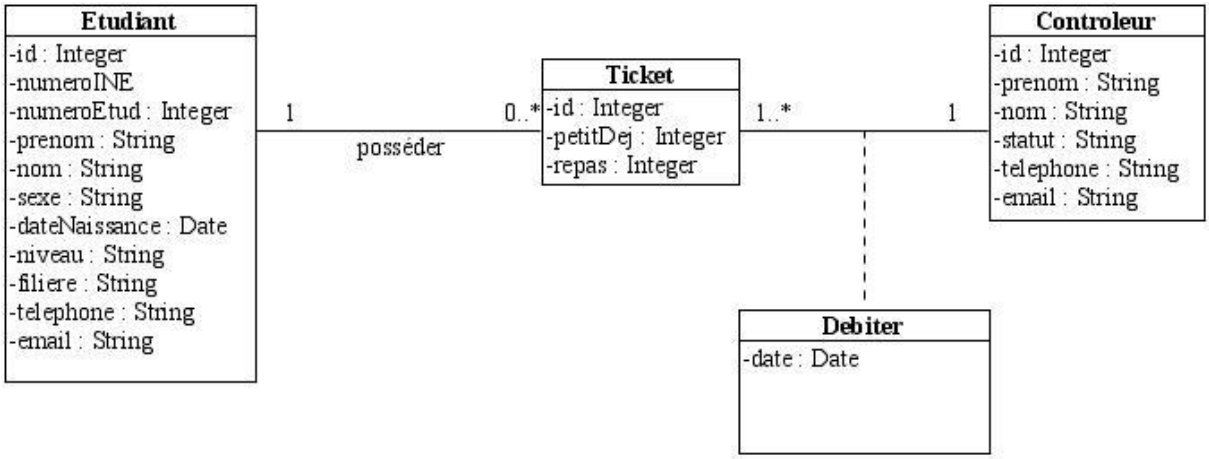


Figure 25 : Diagramme de classes pour le microservice contrôle d'accès au restaurant

Conclusion

Dans cette partie, nous avons développé en détail la conception générale et conception détaillées du système.

La conception générale nous a permis d'évoquer l'architecture du système, les différents diagrammes (composant, package et déploiement). Enfin nous avons terminé par la conception détaillée de l'application dans laquelle nous avons représenté les diagrammes de classe afin de bien préparer la prochaine étape qui est l'implémentation et la présentation de l'application.

CHAPITRE IV : IMPLEMENTATION DU SYSTEME

Dans ce chapitre, nous allons lister l'ensemble des outils de développement du système, présenté les model logique des données (MLD) pour la création des bases de données des différents microservices et enfin expliquer en détail l'implémentation du système.

IV.1 Les outils de développement

Nous avons utilisé plusieurs outils logiciel pour une meilleure réalisation du projet. Ces différents outils nous ont permis de faire l'ensemble des diagrammes présentés dans ce document et d'implémenter les différents programmes informatiques.

Les outils utilisés sont :

- Spring Boot [https://projects.spring.io/spring-boot/] Version: 2.4.x
  Spring Boot est un Framework qui permet de démarrer rapidement le développement d'applications ou services en fournissant les dépendances nécessaires et en auto-configurant celles-ci.
- Spring Cloud [http://projects.spring.io/spring-cloud/] Version : 2020.0.2
  Spring Cloud fournit des outils aux développeurs pour créer rapidement certains des modèles courants dans les systèmes distribués (par exemple, gestion de configuration, découverte de services, routage intelligent).
- Java [http://www.oracle.com/technetwork/java/javase/downloads/indexjsp-138363.html] Version 11
  Java est un langage de programmation populaire et est utilisé à grande échelle dans le monde entier pour le développement d'applications. Il présente des avantages tels que le multithreading, l'extensibilité, la gestion de la mémoire, la haute sécurité, le support communautaire, etc.
- Maven [https://maven.apache.org/] Version : 4.0.0
  Apache Maven est un outil de gestion et d'automatisation de production des projets logiciels Java en général et Java EE en particulier. Il est utilisé pour automatiser l'intégration continue lors d'un développement de logiciel.
- IntelliJ IDEA [https://www.jetbrains.com/idea/download/] Version: 2020.3.2 x64

IntelliJ IDEA est un environnement de développement intégré (IDE) pour les langages JVM conçu pour maximiser la productivité des développeurs

- Visual Paradigm for UML [https://www.visual-paradigm.com/features/] Version: 6.3 Enterprise Edition

  Visual Paradigm for UML est un logiciel permettant aux programmeurs de mettre en place des diagrammes UML.

- Postman [https://www.postman.com/downloads/] Version: 8.9.1

  Postman est une plate-forme d'API pour la création et l'utilisation d'API. Postman simplifie chaque étape du cycle de vie des API et rationalise la collaboration afin de créer de meilleures API, plus rapidement.

- Angular [https://angular.io/guide/setup-local] Version : 11

  Angular est un framework côté client, open source, basé sur TypeScript. C'est une plate-forme pour la création d'applications Web mobiles et de bureau.

- Keycloak [https://www.keycloak.org/downloads] Version : 12.0.4

  Keycloak est une solution Open Source de gestion des identités et des accès pour les applications et services modernes. Il permet d'ajouter l'authentification aux applications et aux services sécurisés avec un minimum d'effort.

IV.2 Création des bases de données des différents microservices

L'architecture en microservices obéit à une règle importante, à savoir que chaque microservice doit posséder les données et la logique de son domaine. Au même titre qu'une application complète, chaque microservice doit posséder sa logique et ses données dans un cycle de vie autonome, avec un déploiement indépendant par microservice.

Le Modèle Logique des Données (MLD) est une étape intermédiaire pour passer du modèle entité-association (EA), qui est un modèle sémantique, vers une représentation physique des données : fichiers, SGBD hiérarchique, SGBD réseau, SGBD relationnel. Pour notre cas nous utilisons un SGBD relationnel.

IV.2.1 Le modèle logique de données du microservice Etudiant

La **figure 26** ci-dessous représente le modèle logique des données du microservice Etudiant pour l'implémentation de sa base de données.

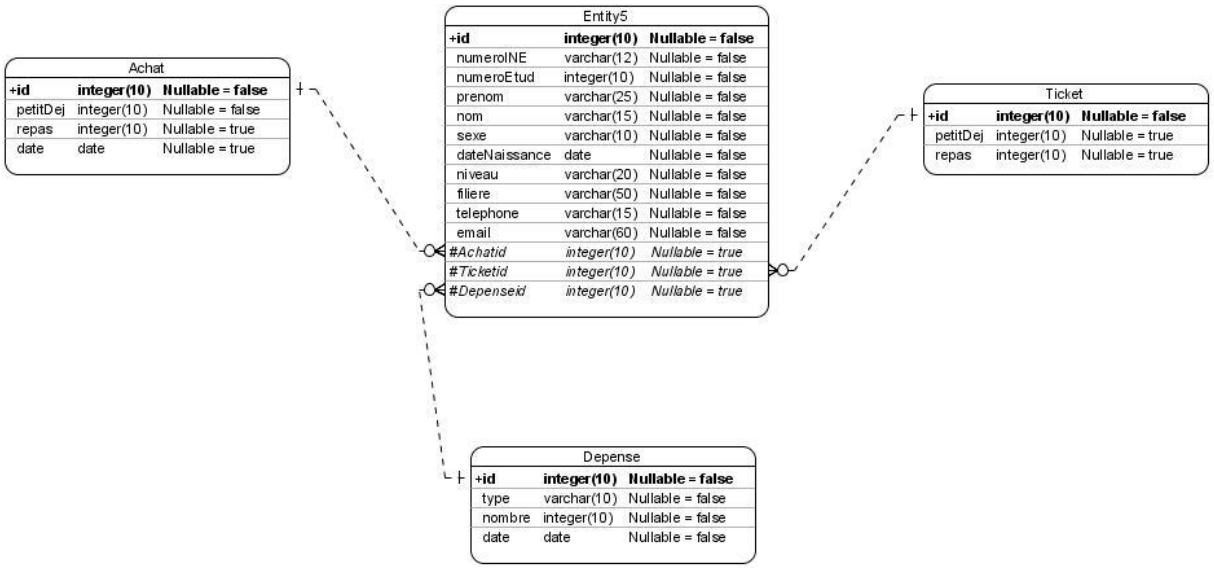


Figure 26 : Modèle logique de données du microservice Etudiant

IV.2.2 Création de la base de données du microservice Etudiant

La **figure 27** ci-dessous représente la création de la base de données du microservice Etudiant.

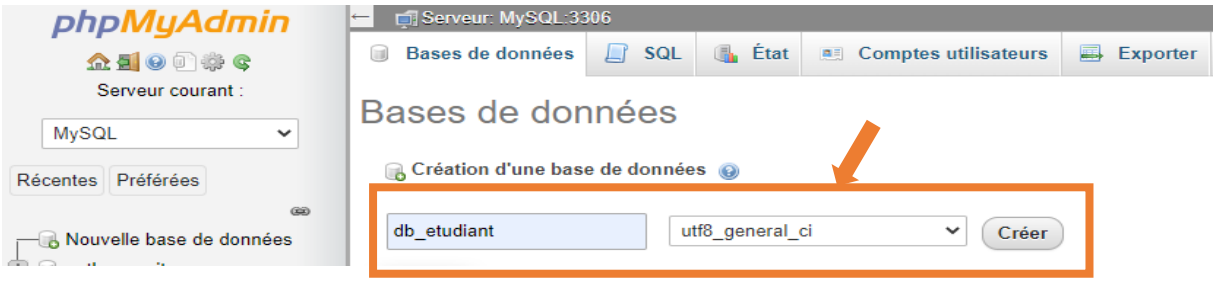


Figure 27 : Base de données du microservice Etudiant

IV.2.3 Le modèle logique de données du microservice Vendeur de ticket

La **figure 28** ci-dessous représente le modèle logique des données du microservice Vendeur de ticket pour l'implémentation de sa base de données.

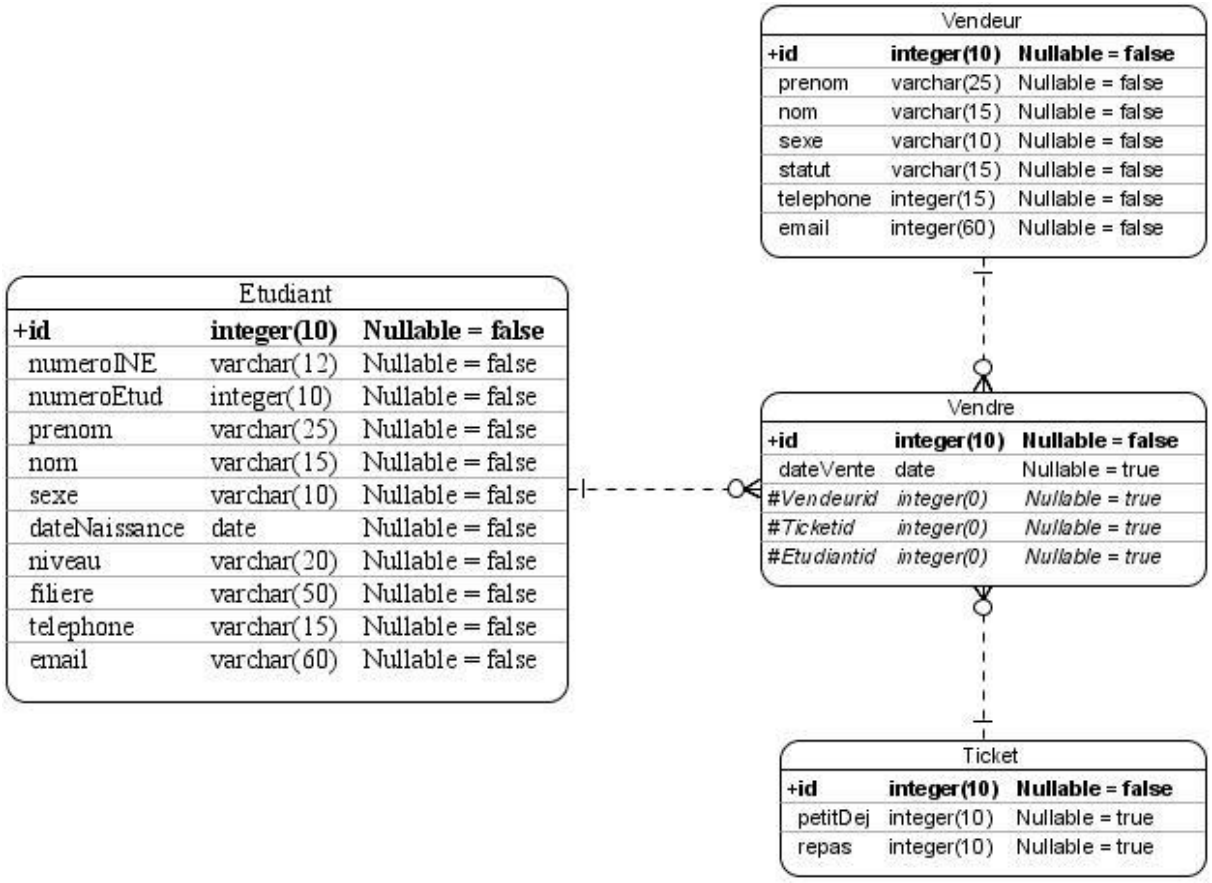


Figure 28 : Modèle logique de données du microservice Vendeur de ticket

IV.2.4 Création de la base de données du microservice Vendeur de ticket

La **figure 29** ci-dessous représente la création de la base de données du microservice Etudiant.

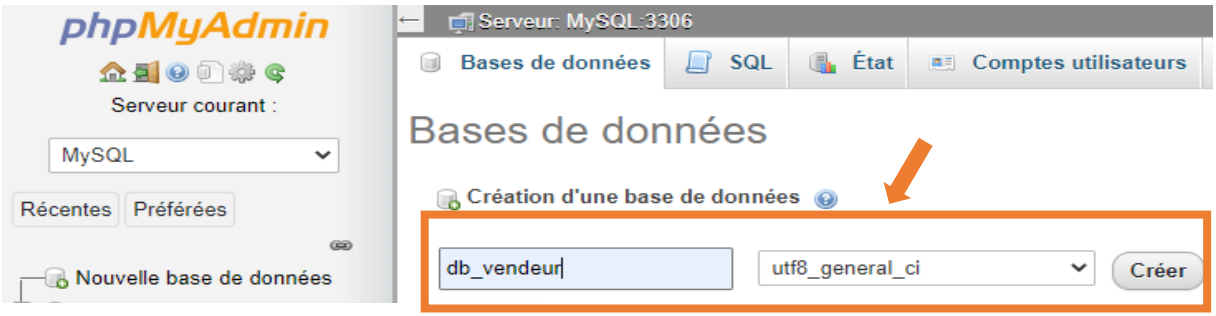


Figure 29 : Base de données du microservice Vendeur

IV.2.5 Le modèle logique de données du microservice Contrôleur ou portier

La **figure 30** ci-dessous représente le modèle logique des données du microservice Etudiant pour l'implémentation de sa base de données.

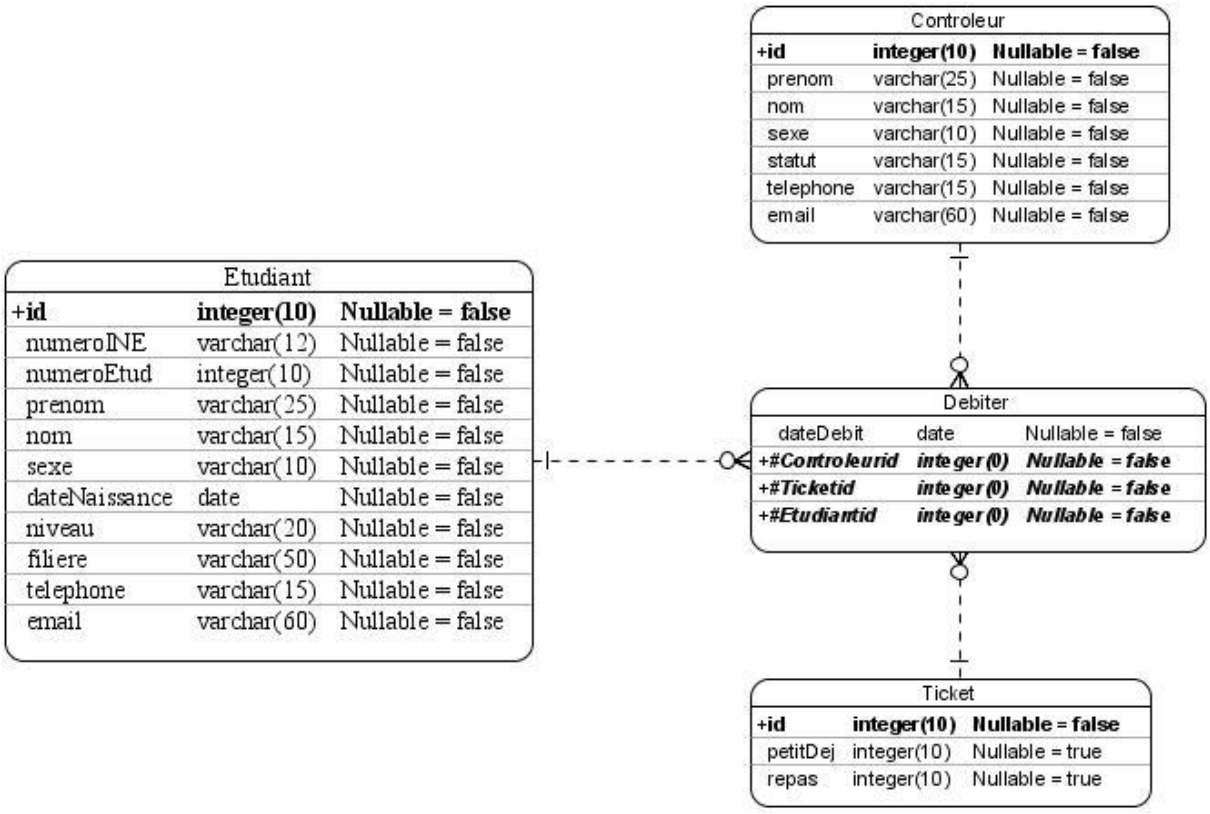


Figure 30 : Modèle logique de données du microservice Contrôleur ou portier

IV.2.6 Création de la base de données du microservice Contrôleur ou portier

La **figure 31** ci-dessous représente la création de la base de données du microservice Etudiant.

Figure 31 : Base de données du microservice Contrôleur ou portier

IV.3 Implémentation des microservices

IV.3.1 Création d'un projet Microservice

Pour la création du projet, nous nous rendons à l'adresse **spring initializr** [https://start.spring.io/] pour le téléchargement du squelette d'un projet spring boot. Il nous permet d'ajouter toutes les dépendances nécessaires au bon fonctionnement du microservice. Les dépendances nécessaires sont :

1. **Spring Web** : Permet de créer des applications Web, y compris RESTful, à l'aide de Spring MVC. Il utilise Apache Tomcat comme conteneur intégré par défaut.
2. **Spring Data JPA** : Permet de conserver les données dans les magasins SQL avec l'API Java Persistance en utilisant Spring Data et Hibernate. Il permet les interactions avec les bases de données relationnelles en s'appuyant sur Hibernate.
3. **Lombok** : Bibliothèque d'annotations Java qui permet de réduire le code passe-partout comme les constructeurs, les getters et setters.

   Exemple :

   `@NoArgsConstructor, @RequiredArgsConstructor et @AllArgsConstructor`

   Constructeurs sur commande : génère des constructeurs qui ne prennent aucun argument, un argument par champ final/non nul ou un argument pour chaque champ.

   `@Data`

   Tous ensemble maintenant : Un raccourci pour `@ToString`, `@EqualsAndHashCode`, `@Getter` sur tous les champs, et `@Setter` sur tous les champs non finaux, et `@RequiredArgsConstructor`

4. **MySQL Driver** : Dépendance vers le pilote de base de données JDBC

La **figure 32** ci-dessous présente la page de téléchargement du projet spring boot avec l'ensemble des dépendances.

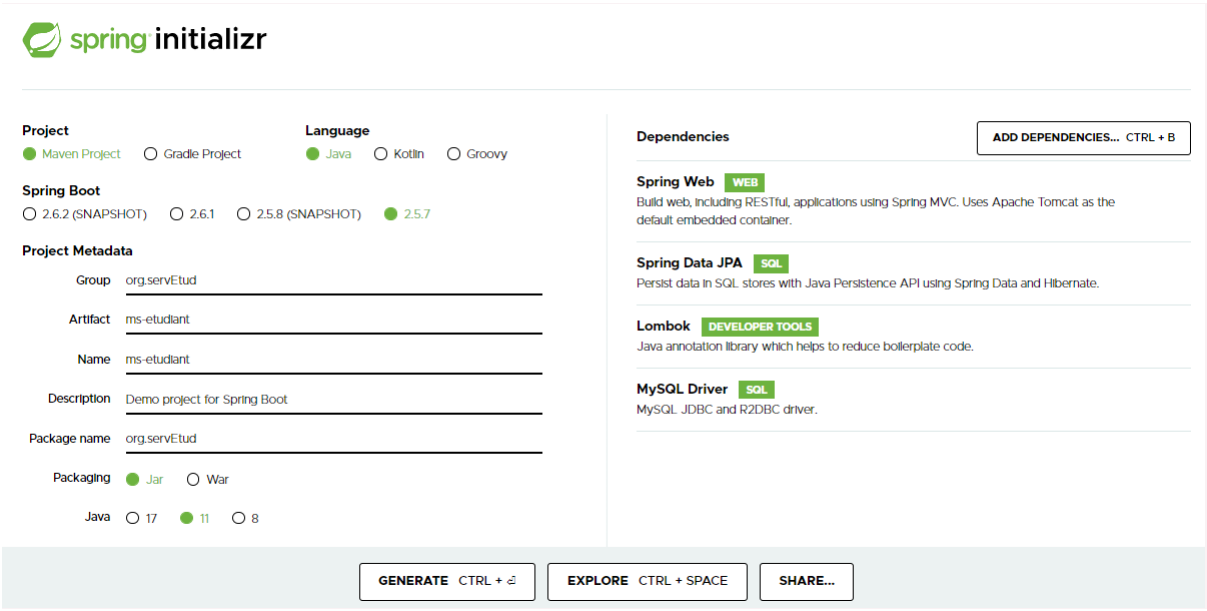


Figure 32 : Création d'un projet Microservice

IV.3.2 Code source

Le code source est structuré en package pour une meilleure structuration de l'ensemble du code. Nous avons séparé notre code comme suite :

- Les codes source de configurations dans le package ***configuration***
- Les codes source qui implémentent les protocoles de sécurité dans le package ***sec***
- Les codes source de l'ensemble des entités dans le package ***entities***
- Les codes source pour l'accès aux couches de données dans le package ***dao***
- Les codes source pour les contrôleurs dans le package ***controller***
- Les codes source qui implémentation l'ensemble des services de notre microservice se trouve dans le package ***service***

La **figure 33** ci-dessous montre la structure du code source du microservice Etudiant.

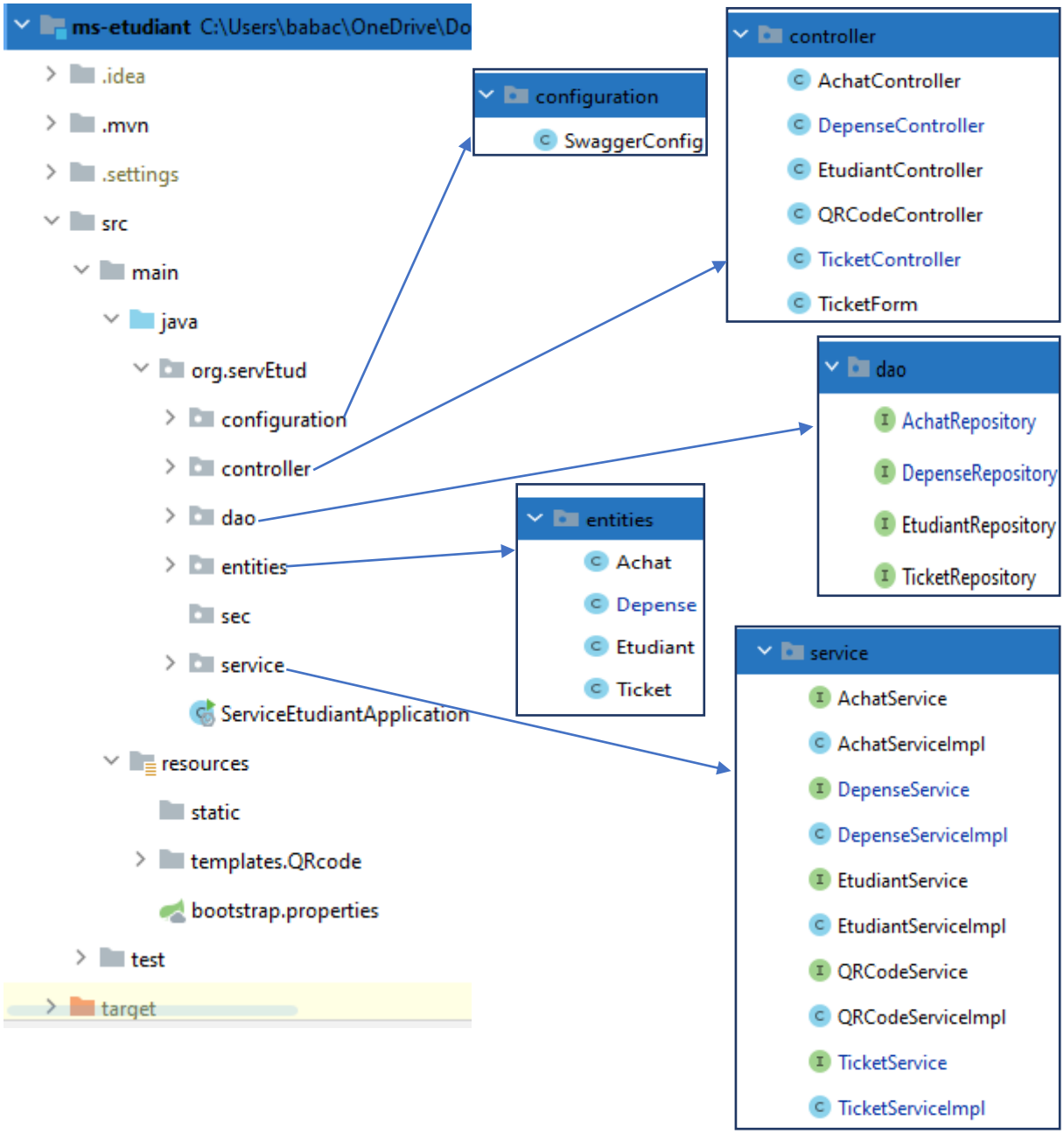


Figure 33 : Structure du code source du microservice Etudiant

IV.3.2.1 Implémentation des entités

En base de données, une entité est un objet du monde réel avec une existence indépendante. L'occurrence d'une entité est un élément particulier correspondant à l'entité et associé à un élément du réel. Chaque entité a des propriétés (ou attributs) qui la décrivent. Chaque attribut est associé à un domaine de valeur.

Lors de l'implémentation nous avons utilisé le principe du mapping Objet/Relationnel (O/R) grâce à **JPA** (Java Persistence API). L'utilisation pour la persistance d'un mapping O/R permet de proposer un niveau d'abstraction plus élevé que la simple utilisation de JDBC : ce mapping permet d'assurer la transformation d'objets vers la base de données et vice versa que cela soit pour des lectures ou des mises à jour (création, modification ou suppression). Nous avons aussi utilisé les annotations de **Lombok** pour la simplification du code source de nos entités.

IV.3.2.1.1 Java Persistence API (JPA)

JPA (Java Persistence API) est un standard de la plateforme JEE qui définit :

- Des règles de correspondance entre objets Java et tables de base de données relationnelles
- Les interfaces (l'API) à respecter.

Les objets principalement manipulés sont les **Entity** : classes persistantes POJO (Plain Old Java Object) avec des annotations JPA définissant les correspondances entre objets/attributs et tables/champs.

❖ **Paramétrage avec Spring Boot**

Par défaut, Spring Boot offre un paramétrage de la persistance avec une configuration respectant les bonnes pratiques en vigueur. Pour affiner ce paramétrage, on peut ajouter des informations dans le fichier *application.properties*

```
spring.datasource.username= root
spring.datasource.password=
spring.datasource.url= jdbc:mysql://localhost:3306/nomBaseDeDonnee
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

❖ Correspondance entre les classes et les tables

La correspondance se fait avec JPA, grâce aux annotations :

@Entity : indique qu'il s'agit d'une entité (classe) persistante

@Table (optionnel) : détermine le nom de la table s'il n'est pas le même que celui de la classe

@Id : indique que l'attribut joue le rôle de clé primaire (simple). Selon la spécification JPA, chaque entité doit avoir un identifiant unique

@GeneratedValue : indique la stratégie de génération de la clé primaire

@Column (optionnel) : détermine le nom de la colonne s'il n'est pas le même que l'attribut de la classe

@OneToMany (1-N)

@ManyToOne (N-1)

@OneToOne (1-1)

@ManyToMany (N-N)

Etc…

Exemple d'implémentation avec les entités du microservice Etudiant

- **Entité Etudiant**

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Etudiant {
   @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   @Column(unique = true)
   private String numeroEtud;
   private String prenom;
   private String nom;
   private String sexe;
   private Date dateNaissance;
   private String niveau;
   private String filiere;
   @Column(unique = true)
   private String telephone;
   @Column(unique = true)
   private String email;
   @JsonManagedReference
   @OneToMany(mappedBy="etudiant")
   private Collection<Ticket> tickets;
   @JsonManagedReference
   @OneToMany(mappedBy="etudiant")
```

```
    private Collection<Depense> depenses;
    @JsonManagedReference
    @OneToMany(mappedBy="etudiant")
    private Collection<Achat> achats;
}
```

- **Entité Ticket**

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Ticket {
    @Id
    private Long id;
    private int petitDej;
    private int repas;
    @JsonBackReference
    @ManyToOne
    @JoinColumn
    private Etudiant etudiant;
}
```

- **Entité Achat**

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Achat {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private int petitDej;
    private int repas;
    private Date petit-déj;
    private String fournisseur;
    @JsonBackReference
    @ManyToOne
    @JoinColumn
    private Etudiant etudiant;
}
```

- **Entité Depense**

```
    @Data
    @NoArgsConstructor
    @AllArgsConstructor
    @Entity
    public class Depense {
        @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
```

```
    private String type;
    private int nombre;
    private Date date;
    @JsonBackReference
    @ManyToOne
    @JoinColumn
    private Etudiant etudiant;
}
```

IV.3.2.2 Implémentation de la couche d'accès aux données

L'implémentation de la persistance doit être isolée dans la couche d'accès aux données. Elle s'appuie sur le design pattern DAO (Data Access Object) pour rendre la couche métier indépendante de la source de données sous-jacente (SQL, mapping O/R.…).

Le DAO propose généralement les méthodes **CRUD (Create/Read/Update/Delete)** suivantes:

- **get** : récupération d'un objet persistant grâce à son identifiant
- **create** : enregistrement d'un nouvel objet persistant
- **update** : mise à jour d'un objet persistant
- **delete** : suppression d'un objet persistant
- **findAll** : récupération de toutes les instances d'une classe persistante
- **findBy** : récupération des objets persistants selon des critères de sélection ou de tri

Spring Data fournit une interface générique **Repository<*T, ID*>**, permettant de spécifier :

- Le type d'entité (**T**) géré par ce repository
- Le type de l'identifiant de cette entité (**ID**).

Pour accéder aux données, il suffit de créer une interface annotée ***@Repository***, qui étend ***JpaRepository<T, ID>***.

Ainsi, on peut :

- Bénéficier automatiquement des méthodes héritées (ex : CRUD)
- Ajouter de nouvelles signatures de méthodes respectant un nommage précis :
    - FindByXxx... (Xxx étant le nom d'un attribut de la classe persistante)
- Ajouter des **query methods** annotées ***@Query***, en précisant la requête SQL à exécuter [11]

Exemple d'implémentation avec les repository du microservice Etudiant

- **EtudiantRepository**

```
@Repository
public interface EtudiantRepository extends JpaRepository<Etudiant, Long> {

   Etudiant findByNumeroEtud(String numeroEtud);

   Etudiant findByEmail(String email);

   @Query(value = "SELECT * FROM etudiant WHERE id=:id", nativeQuery=true)
   Etudiant findEtudiantById(@Param("id") Long id);

}
```

- **TicketRepository**

```
@Repository
public interface TicketRepository extends JpaRepository<Ticket, Long> {

   // Mettre à jour le nombre de ticket d'un étudiant
   @Transactional
   @Modifying
   @Query(value = "UPDATE ticket SET petit_dej=petit_dej+pd, repas=repas+rp
WHERE etudiant_id=id", nativeQuery=true)
   void updateTicket(@Param("id") String id, @Param("pd") int pd,
@Param("rp") int rp);

   // Vérifier si l'étudiant à une fois fait un achat de ticket
   @Query(value = "SELECT * FROM ticket WHERE etudiant_id=:id",
nativeQuery=true)
   List<Ticket> mesTickets(@Param("id") String id);

}
```

- **AchatRepository**

```
@Repository
public interface AchatRepository extends JpaRepository<Achat, Long> {

    @Query(value = "SELECT id, date, petit_dej, repas, etudiant_id,
fournisseur FROM achat " +
            "WHERE etudiant_id=:etudiantId ORDER BY date DESC LIMIT 5",
            nativeQuery = true)
    List<Achat> findAchatByEtudiant(@Param("etudiantId") Long etudiantId);
}
```

- **DepenseRepository**

```
@Repository
public interface DepenseRepository extends JpaRepository<Depense, Long> {

    @Query(value = "SELECT id, date, nombre, type, etudiant_id FROM"+
"depense WHERE etudiant_id=:etudiantId ORDER BY date DESC LIMIT 5",
nativeQuery = true)
    List<Depense> findDepensesByEtudiant(@Param("etudiantId") Long
etudiantId);
}
```

IV.3.2.3 Implémentation des services

Les composants de service sont les fichiers de classe qui contiennent l'annotation *@Service*. Ces fichiers de classe sont utilisés pour écrire la logique métier dans une couche différente, séparée du fichier de classe *@RestController*.

@Service annote les classes au niveau de la couche de service.

En plus d'être utilisé dans la couche de service, il n'y a pas d'autre utilisation spéciale pour cette annotation. La logique de création d'un fichier de classe de composant de service est illustrée par une interface et la classe qui l'implémente contient l'annotation ***@Service***.

Exemple d'implémentation avec le microservice Etudiant

```
public interface TicketService {

   List<Ticket> MesTickets(String id);

   Ticket updateNbrTicket(Long idEtudiant, int petitDej, int repas);

   int getNbrPetitDej(Long idEtudiant);

   int getNbrRepas(Long idEtudiant);

   void updateNbrPetitDej(Long idEtudiant);

   void updateNbrRepas(Long idEtudiant);
}


@Service
@Transactional
public class TicketServiceImpl implements TicketService{

   @Autowired
   private TicketRepository ticketRepository;

   @Autowired
   private EtudiantService etudiantService;
```

```
    @Override
    public List<Ticket> MesTickets(String id) {
        var tickets = (List<Ticket>) ticketRepository.mesTickets(id);
        return tickets;
    }

    @Override
    public Ticket updateNbrTicket(Long idEtudiant, int petitDej, int repas)
{

        Ticket updateTicket;

        if (ticketRepository.findById(idEtudiant).isPresent()) {

            Ticket ticket = ticketRepository.findById(idEtudiant).get();

            ticket.setPetitDej(petitDej + ticket.getPetitDej());
            ticket.setRepas(repas + ticket.getRepas());

            updateTicket = ticketRepository.save(ticket);

        } else {

            Ticket ticket = new Ticket();

            ticket.setId(idEtudiant);
            ticket.setPetitDej(petitDej);
            ticket.setRepas(repas);
            ticket.setEtudiant(etudiantService.findEtudiantById(idEtudiant));

            updateTicket = ticketRepository.save(ticket);
        }
        return new Ticket(updateTicket.getId(), updateTicket.getPetitDej(),
updateTicket.getRepas(), updateTicket.getEtudiant());
    }
}
```

IV.3.2.4 Implémentation des contrôleurs

Un contrôleur est une classe portant l'annotation ***@Controller***. De manière générale, l'objectif d'un contrôleur est de réagir à une interaction avec l'utilisateur. Pour une application Web, cela signifie que l'utilisateur envoie une requête HTTP au serveur.

Une classe annotée avec ***@RestController*** indique qu'il s'agit d'un contrôleur spécialisé pour le développement d'API Web qui est notre cas pour ce projet. ***@RestController*** est simplement une annotation qui regroupe ***@Controller*** et ***@ResponseBody***. Il s'agit donc d'un contrôleur dont les méthodes retournent par défaut les données à renvoyer au client plutôt qu'un identifiant de vue.

Spring MVC a rendu l'écriture de classes et de méthodes de contrôleur de gestionnaire de requêtes très facile. On ajoute simplement quelques annotations comme ***@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping*** et ***@RequestMapping***.

Le **tableau 6** ci-dessous fais la description fonctionnelle de ces annotations.

Tableau 6 : Description des annotations pour les méthodes d'un contrôleur

Annotation	**Type HTTP**	**Objectif**
@GetMapping	GET	Pour la **lecture** de données.
@PostMapping	POST	Pour **l'envoi** de données. Cela sera utilisé par exemple pour créer un nouvel élément.
@PatchMapping	PATCH	Pour **la mise à jour partielle** de l'élément envoyé.
@PutMapping	PUT	Pour **le remplacement complet** de l'élément envoyé.
@DeleteMapping	DELETE	Pour la **suppression** de l'élément envoyé.
@RequestMapping		Intègre tous les types HTTP. Le type souhaité est indiqué comme attribut de l'annotation. Exemple : *@RequestMapping(method = RequestMethod.GET)*

En résumé

- Notre entité du model est modélisée, et ***@Entity*** est l'annotation obligatoire !
- La communication aux données s'effectue via une classe annotée ***@Repository***.
- La classe annotée ***@Service*** se charge des traitements métiers.
- Nos contrôleurs ***@RestController*** nous permettent de définir des URL et le code à exécuter quand ces dernières sont requêtées. [12]

Exemple d'implémentation d'un contrôleur avec le microservice Etudiant

- **Implémentation du contrôleur TicketController**

On a injecté une instance de `ticketService`. Cela permettra d'appeler ses méthodes pour communiquer avec la base de données.

```
@CrossOrigin("http://localhost:4200")
@RestController
public class TicketController {

    @Autowired
    private TicketService ticketService;


   // Mettre à jour le nombre de tickets de l'étudiant
    @PutMapping("nbTickets/update/{petitDej}/{repas}/{id}")
    public void updateNbrTickets(@PathVariable int petitDej, @PathVariable
int repas, @PathVariable Long id){
       ticketService.updateNbrTicket(id, petitDej, repas);
    }

    // Voir le nombre de tickets d'un etudiant par son id
    @GetMapping("tickets/etudiant/{id}")
    public ResponseEntity<List<Ticket>> getTicketByIdEtudiant(@PathVariable
String id){
      return new ResponseEntity<>(ticketService.MesTickets(id),
HttpStatus.OK);
    }

   // Voir le nombre de petit dejeuner d'un etudiant par son id
   @GetMapping("ticket/petitDej/{idEtudiant}")
   public  int getNbrPetitDej(@PathVariable Long idEtudiant){
      return ticketService.getNbrPetitDej(idEtudiant);
   }

   // Voir le nombre de repas d'un etudiant par son id
   @GetMapping("ticket/repas/{idEtudiant}")
   public  int getNbrRepas(@PathVariable Long idEtudiant){
      return ticketService.getNbrRepas(idEtudiant);
   }

   // Mettre à jour le nombre de petit dejeuner d'un etudiant par son id
   @PutMapping("ticket/petitDej/{idEtudiant}/update")
   public void updateNbrPetitDej(@PathVariable Long idEtudiant){
      ticketService.updateNbrPetitDej(idEtudiant);
   }

   // Mettre à jour le nombre de repas d'un etudiant par son id
   @PutMapping("ticket/repas/{idEtudiant}/update")
   public void updateNbrRepas(@PathVariable Long idEtudiant){
      ticketService.updateNbrRepas(idEtudiant);
   }

}
```

IV.3.2.5 Documentation des APIs avec Swagger

Swagger est un projet open source utilisé pour décrire et documenter les API RESTful. Swagger est indépendant du langage et est extensible dans de nouvelles technologies et protocoles au-delà de HTTP.

Spring Boot rend le développement de services RESTful facile et l'utilisation de Swagger facilite la documentation de nos services RESTful.

La création d'une couche d'API back-end introduit un tout nouveau domaine de défis qui va au-delà de la mise en œuvre de simples points de terminaison. Nous avons maintenant des clients qui utiliseront désormais notre API. Nos clients devront savoir comment interagir avec notre API.

Spring Boot et Swagger 2 jouent très bien ensemble. On ajoute simplement les dépendances, un fichier de configuration et un ensemble d'annotations.[13]
Il y'a trois étapes pour intégrer swagger dans un projet spring.

- **Première étape :** On ajoute les dépendances suivantes dans notre *pom.xml* :

```
<dependencies>
    <dependency>
       <groupId>io.springfox</groupId>
       <artifactId>springfox-swagger2</artifactId>
       <version>3.0.0</version>
    </dependency>

    <dependency>
       <groupId>io.springfox</groupId>
       <artifactId>springfox-boot-starter</artifactId>
       <version>3.0.0</version>
    </dependency>

    <dependency>
       <groupId>io.springfox</groupId>
       <artifactId>springfox-swagger-ui</artifactId>
       <version>2.9.2</version>
    </dependency>
</dependencies>
```

- **Deuxième étape :** Ajouter l'annotation `@EnableSwagger2` à notre fichier *nomMicroserviceApplication.java*

```
@EnableSwagger2
public class nomMicroserviceApplication {
        // reste du code
}
```

- **Troisième étape : Configuration Java**

La configuration de Swagger est principalement centrée sur le bean ***Docket*** :

- On a d'abord créé le package « configuration » et y créé la classe SwaggerConfig.java et y ajouter les annotations **@EnableSwagger2 et @Configuration**

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
          .apis(RequestHandlerSelectors.basePackage("org.servEtud"))
                .paths(PathSelectors.any())
                .build();
    }
}
```

Dans cette classe de configuration, l'annotation `@EnableSwagger2` active la prise en charge de Swagger dans la classe. La méthode `select()` appelée sur l'instance du bean Docket renvoie un `ApiSelectorBuilder`, qui fournit les méthodes `api()` et `paths()` qui sont utilisées pour filtrer les contrôleurs et les méthodes qui sont documentés à l'aide de prédicats String.

Dans le code, le prédicat `RequestHandlerSelectors.basePackage` correspond au `org.servEtud` package de base qui contient notre code du microservice Etudiant pour filtrer l'API. [13]

Nous pouvons ainsi ajouter une description pour chaque API grâce à l'annotation `@Api`, comme illustré ci-après :

```
@RestController
@Api(value = "TicketController", description="API pour es opérations CRUD
sur les produits.")
public class TicketController {
    // Reste du code du controleur
}
```

La **figure 34** ci-dessous montre La documentation générée par Swagger-UI

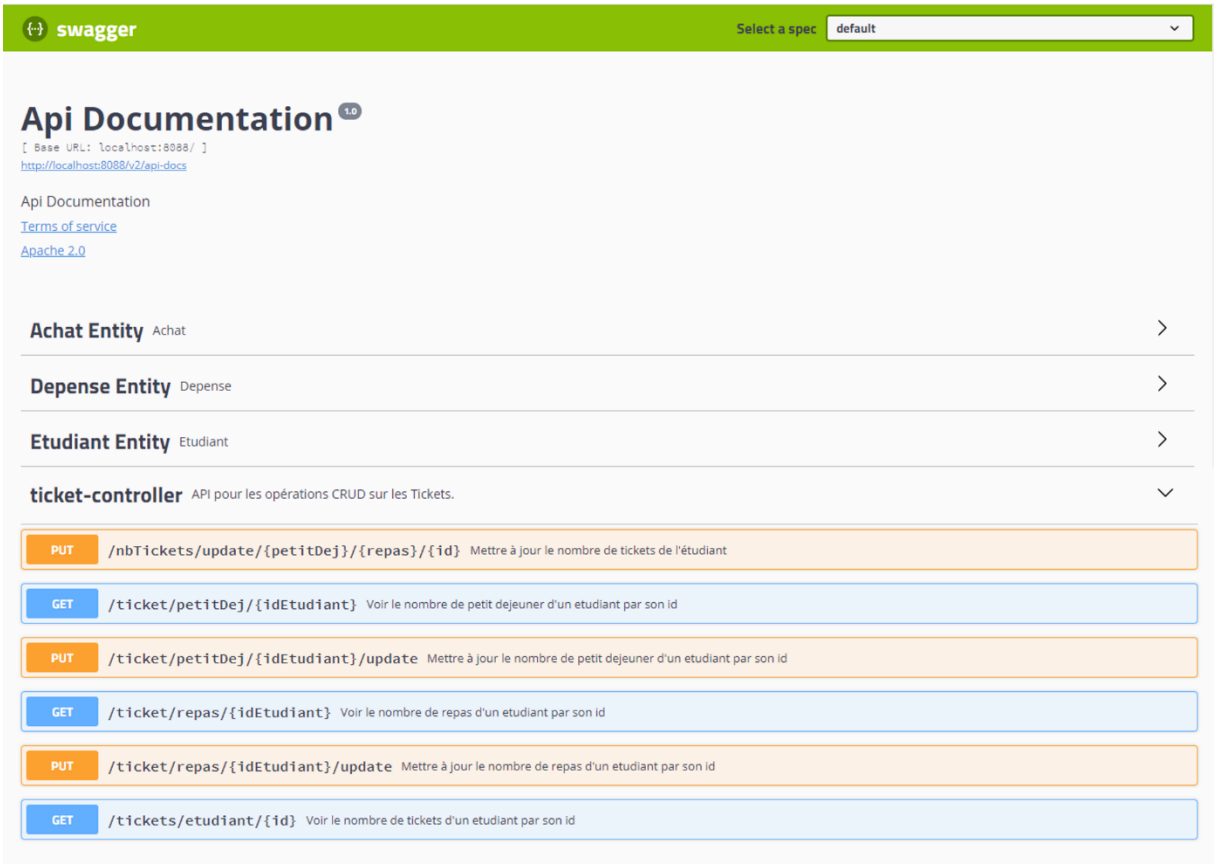


Figure 34 : documentation générée par Swagger-UI

On a aussi utilisé l'annotation `@ApiModelProperty` pour décrire les propriétés des modèles. Avec `@ApiModelProperty`, on peut également documenter une propriété.

Le code de notre **classe Ticket.java du microservice Etudiant** est le suivant.

```
@Entity
public class Ticket {
   @Id
   @ApiModelProperty(notes = "ID du ticket auto_increment", required=true)
   private Long id;
   @ApiModelProperty(notes = "Le nombre de petit déjeuner")
   private int petitDej;
   @ApiModelProperty(notes = "Le nombre de repas")
   private int repas;
   @JsonBackReference
   @ManyToOne
   @JoinColum
   @ApiModelProperty(notes="L'étudiant à qui le ticket appartient",
required=true)
   private Etudiant etudiant;
}
```

La documentation générée par Swagger 2 pour la classe Ticket est représenté par la **figure 35** suivante :

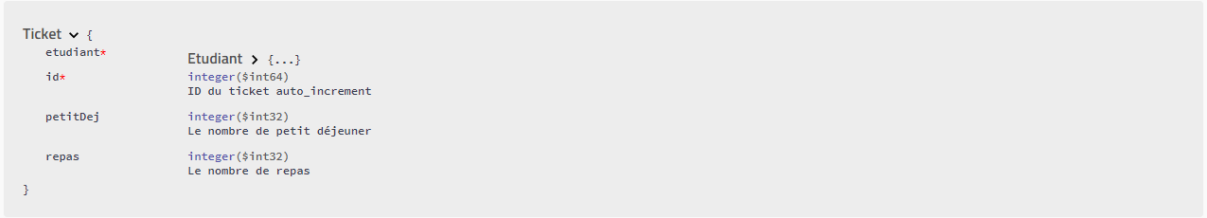


Figure 35 : Documentation générée par Swagger 2 pour la classe Ticket

IV.3.2.6 Implémentation des Edge microservices

Les **Edge Microservices** sont des **Microservices** spécialisés dans l'orchestration des **Microservices** centraux responsables de la logique de l'application. Ils permettront à notre application d'être 100% compatible avec le cloud.

Les Edge microservices implémentés pour ce projet sont : ***Config Server, Discovery server* et *Proxy Server*.**

- **Implémentation du microservice *Config Server***

Pour ce microservice, spring boot nous offre la dépendance **spring cloud config** qui gère la Gestion centralisée pour la configuration via Git, SVN ou HashiCorp Vault. Pour notre cas nous utilisons Git.

La **figure 36** ci-dessous illustre la création du *microservice config server*

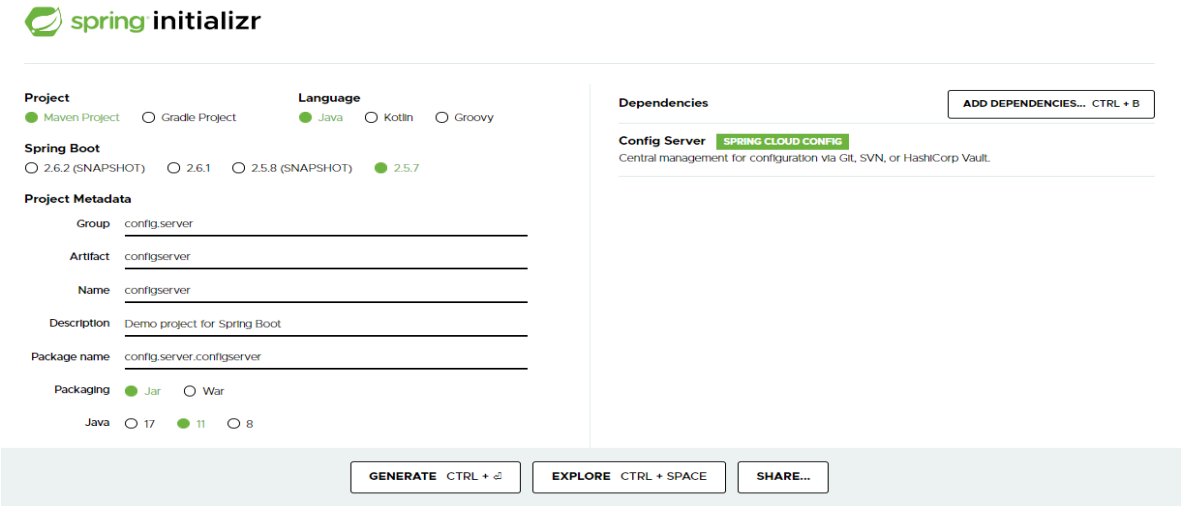


Figure 36 : Création du microservice Config Server

Après téléchargement du projet, pour finaliser la configuration nous devons d'abord ajouter l'annotation `@EnableConfigServer` au-dessus de la classe main **ConfigserverApplication.java** illustré ci-dessous.

```
@SpringBootApplication
@EnableConfigServer
public class ConfigserverApplication {
   public static void main(String[] args) {
      SpringApplication.run(ConfigserverApplication.class, args);
   }
}
```

Ensuite on doit renseigner les paramètre suivant dans ***application.properties***

```
spring.application.name=configServer
server.port:8084
management.server.servlet.context-path=/configserver
spring.cloud.config.server.git.uri = https://github.com/Babacar-Diago/configCloud.git
```

Lien du git distant contenant l'ensemble des fichiers de configuration du système

❖ Implémentation du microservice *Discovery server*

Pour ce microservice, spring boot nous offre la dépendance **Spring Cloud Discovery (Eureka),** **Eureka** va permettent l'enregistrement des instances de Microservices en vue d'être découvertes par d'autres Microservices. Nous devons aussi ajouter la dépendance **Config Client** qui permet aux clients de se connecter au serveur Spring Cloud Config pour récupérer la configuration de l'application.

La **figure 37** ci-dessous illustre la création du *microservice Discovery server*

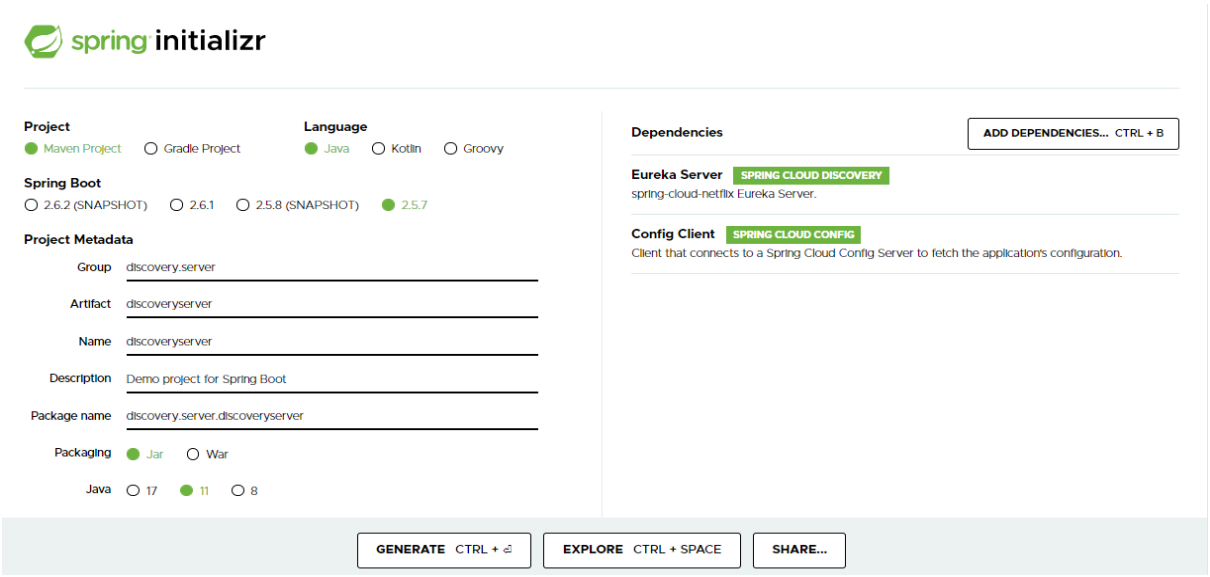


Figure 37 : Création microservice Discovery server

Après téléchargement du projet, pour finaliser la configuration nous devons d'abord ajouter l'annotation `@EnableEurekaServer` au-dessus de la classe main **DiscoveryserverApplication.java** illustré ci-dessous.

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryserverApplication {

   public static void main(String[] args) {
      SpringApplication.run(DiscoveryserverApplication.class, args);
   }

}
```

Ensuite nous devons renommer ***application.properties*** en ***bootstrap.properties*** et y renseigner les paramètre suivant.

```
spring.application.name=discoveryserver

spring.cloud.config.uri = http://localhost:8084
```

uri vers le microservice config server pour récupérer sa configuration

- **Implémentation du microservice *Proxy server***

Pour ce microservice, nous devons ajouter la dépendance suivante dans le fichier **pom.xml**

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
    <version>2.1.3.RELEASE</version>
</dependency>
```

Nous devons aussi ajouter les dépendances **Config Client** et **Eureka Discovery Client**

- *Config Client* permet aux clients de se connecter au serveur Spring Cloud Config pour récupérer la configuration de l'application.
- *Eureka Discovery Client* permet à un service basé sur REST pour localiser des services à des fins d'équilibrage de charge et de basculement des serveurs de niveau intermédiaire.

LA **figure 38** ci-dessous illustre la création du *microservice Proxy server*

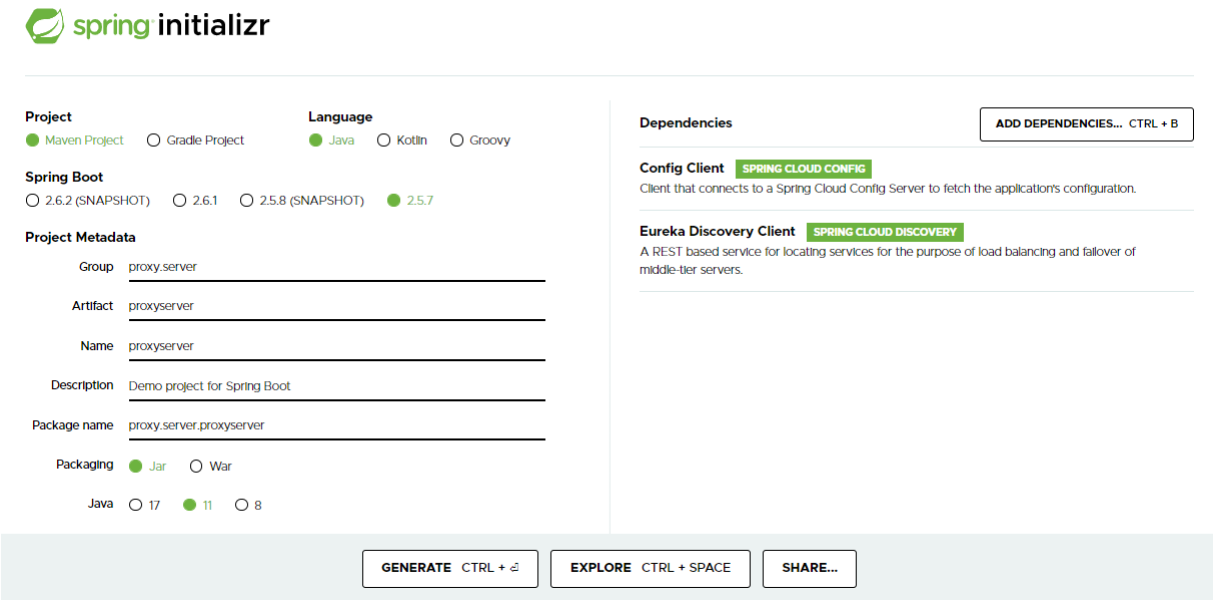


Figure 38 : Création du microservice Proxy server

Après téléchargement du projet, pour finaliser la configuration nous devons d'abord ajouter l'annotation `@EnableZullProxy` au-dessus de la classe main **ProxyserverApplication.java** illustré ci-dessous.

```
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ProxyserverApplication {
   public static void main(String[] args) {
      SpringApplication.run(ProxyserverApplication.class, args);
   }
}
```

Ensuite nous devons renommer ***application.properties*** en ***bootstrap.properties*** et y renseigner les paramètre suivant.

```
spring.application.name=proxyserver
spring.cloud.config.uri=http://localhost:8084
```

Il est a noté aussi que les fichiers de configuration des microservices *Discovery Server* et *Proxy Server* sont externalisés respectivement dans *discoveryserver.properties* et *proxyserver.properties*. Ces deux derniers sont dans le repo distant.

IV.1.3.2.7 Intégration des Edge microservices dans les autres microservices

Dans le chapitre 3 nous avons présenté l'architecture globale de notre application. Nous allons dans cette partie finalisé cette architecture en permettant aux autres microservices d'interagir avec les Edge microservices.

Pour cette intégration, nous allons procéder par trois étapes :

- **Première étape** : Centraliser les configurations de nos Microservices
- **Deuxième étape :** Ajouter les dépendances nécessaires à notre *pom.xml*
- **Troisième étape :** Ajouter une annotation à notre fichier *nomMicroserviceApplication.java*

Première étape : Centraliser les configurations de nos Microservices

1. On créer un fichier *.properties* portant le même nom que notre Microservices et y mettre toute votre configuration.
    a. Dans le fichier de configuration globale on doit ajouter la ligne ci-dessous :

```
eureka.client.serviceUrl.defaultZone:http://localhost:8085/discoveryserver/eureka/
```

Cette ligne permet à ce microservice de pouvoir accéder au microservice discoveryServer afin d'inscrire l'instance qui est démarrée pour qu'il puisse être découvert par les autres microservices

    b. Toujours dans le fichier de configuration globale, on doit ensuite ajouter le reste de la configuration du microservice

2. On renomme *application.properties* en *bootstrap.properties*
3. Dans *bootstrap.properties* on met le code suivant :

```
spring.application.name=nom_du_microservice

spring.cloud.config.uri= http://localhost:8084
```

Cette ligne permet à ce microservice de pouvoir accéder au microservice configServer afin de récupérer sa configuration globale

Deuxième étape : Ajouter les dépendances nécessaires à notre *pom.xml*

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-config</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-tomcat</artifactId>
   <scope>provided</scope>
</dependency>
```

Troisième étape : Ajouter l'annotation `@EnableDiscoveryClient` à notre fichier *nomMicroserviceApplication.java*

```
@EnableDiscoveryClient
public class nomMicroserviceApplication {
        // reste du code
}
```

Conclusion

Dans ce chapitre, nous avons pu présenter les outils utilisés pour la réalisation du projet. Nous avons aussi présenté certains code source de l'application. Le chapitre qui suit présente les différentes interfaces de l'application.

CHAPITRE V : PRESENTATION DE L'APPLICATION

Dans cette partie, nous faisons une présentation des fonctionnalités de cette application. Les interfaces graphiques sont développées avec Angular qui est un Framework JavaScript. C'est un framework côté client, open source, basé sur TypeScript, et co-dirigé par l'équipe du projet « Angular » à Google et par une communauté de particuliers et de sociétés. Notre application est responsive donc s'adapte aux différents types d'écran.

V.1 L'interface d'inscription d'un étudiant

L'interface d'inscription permet à l'étudiant de s'inscrire sur la plateforme et de pouvoir accéder à son compte pour bénéficier de toutes les fonctionnalités développées pour les étudiants. La **figure 39** illustre la page d'inscription des étudiants.

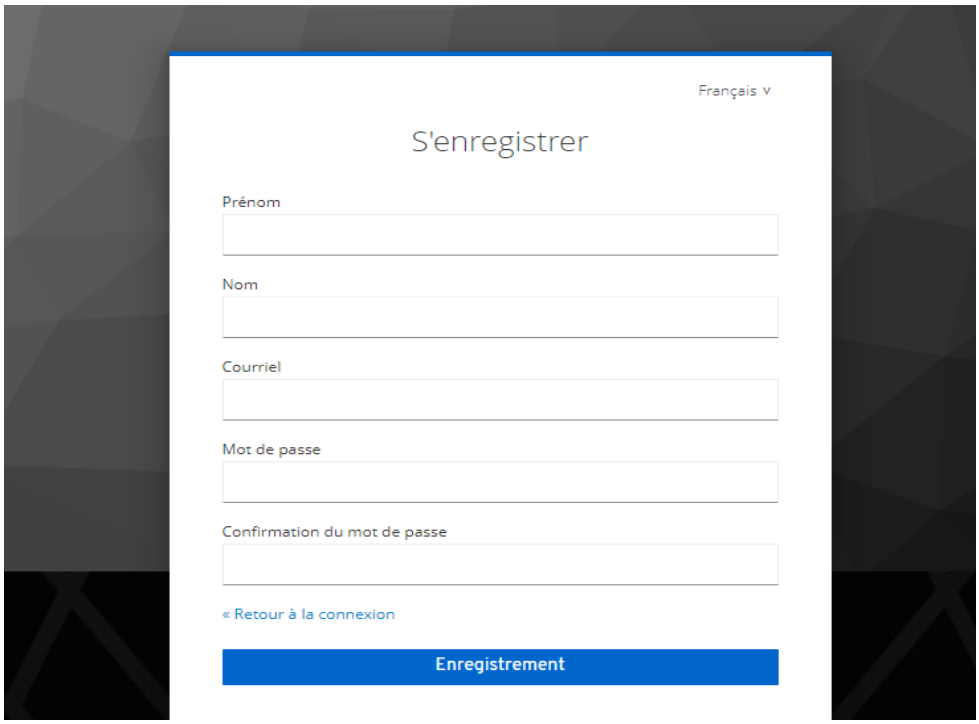


Figure 39 : Page d'inscription des étudiants

V.2 L'interface de connexion

Cette interface permet aux différents utilisateurs de se connecter. Si l'utilisateur donne un login et un mot de passe correct, il accède à sa page d'accueil, sinon un message d'erreur lui est envoyé. La **figure 40** ci-dessous illustre la page de connexion de notre application.

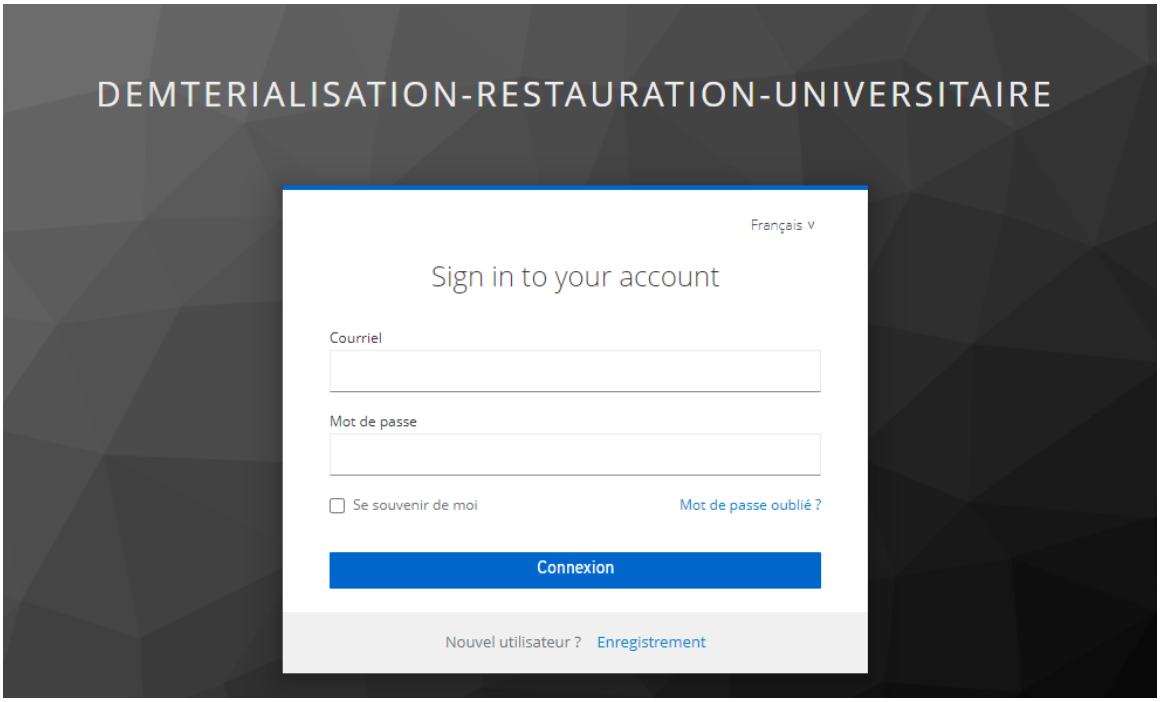


Figure 40 : Interface de connexion

V.3 Les interfaces pour l'étudiant

L'étudiant bénéficie de quatre pages dont une page d'accueil, une page lui permettant d'acheter un ticket en ligne, une page lui permettant de consulter l'historique de ses achats des tickets et une page pour consulter l'historique de ses entrées au restaurant universitaire.

V.3.1 L'interface page d'accueil

Cette page contient plusieurs informations dont le nombre de ticket disponible de l'étudiant, le code QR permettant de l'identifier et l'historique de ses achats et entrées au restaurant. La **figure 41** ci-dessous présente cette page d'accueil.

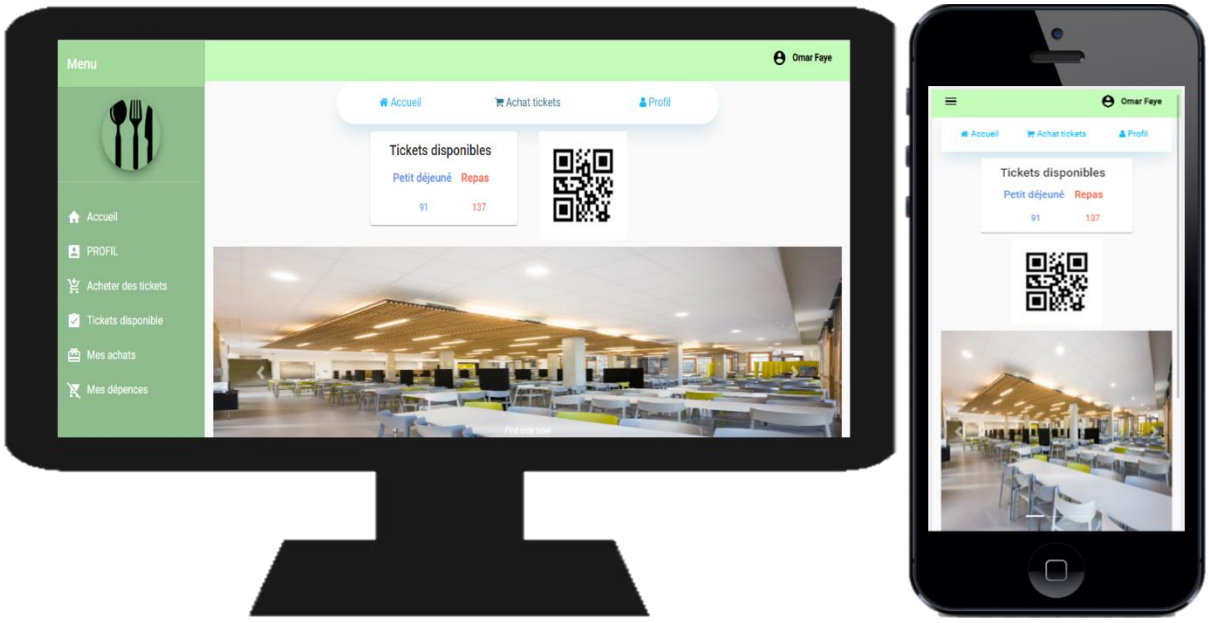


Figure 41 : Page d'accueil de l'étudiant

V.3.2 L'interface pour acheter des tickets de restaurant

Cette interface permet à l'étudiant d'acheter des tickets de petit déjeuner et de repas. L'étudiant pourra payer ses tickets en ligne grâce Univ-Money. La **figure 42** permet d'illustrer cette interface.

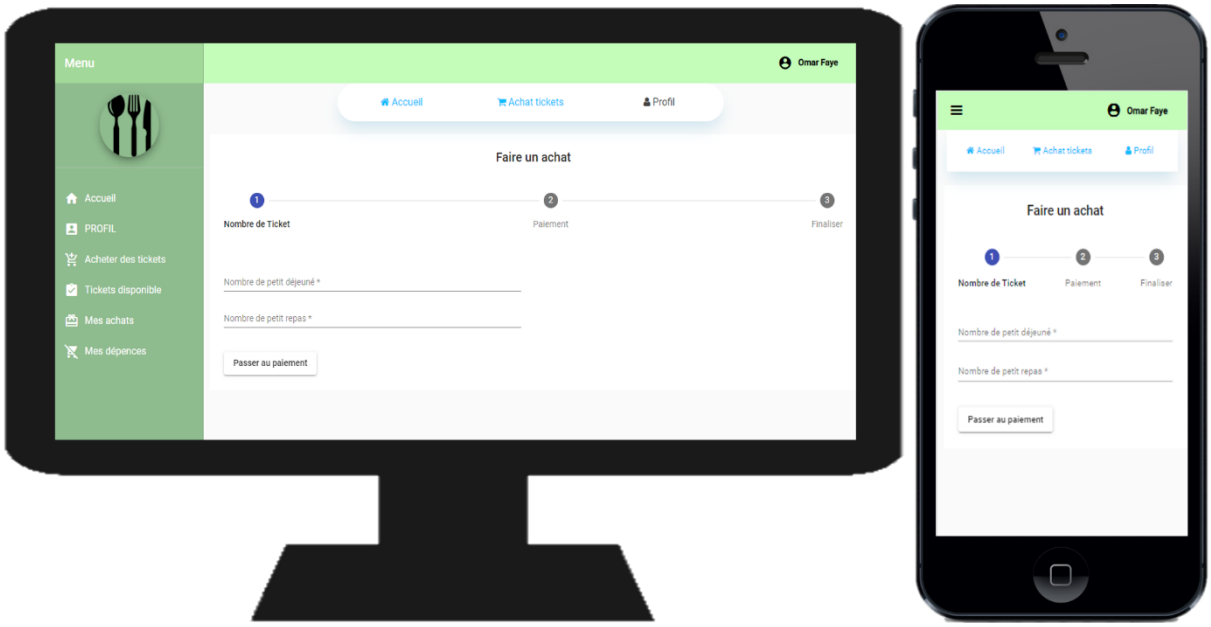


Figure 42 : Page pour achats de ticket

V.3.3 L'interface pour consulter l'historique de ses achats

Cette interface permet à l'étudiant de consulter l'historique l'ensemble de ses achats de ticket. La **figure 43** ci-dessous illustre cette page.

Figure 43 : Historique de l'ensemble de ses achats de ticket de l'étudiant

V.3.4 L'interface pour consulter l'historique de ses entrées au restaurant

Cette interface permet à l'étudiant de consulter l'historique l'ensemble de ses entrées au restaurant. La **figure 44** ci-dessous illustre cette page.

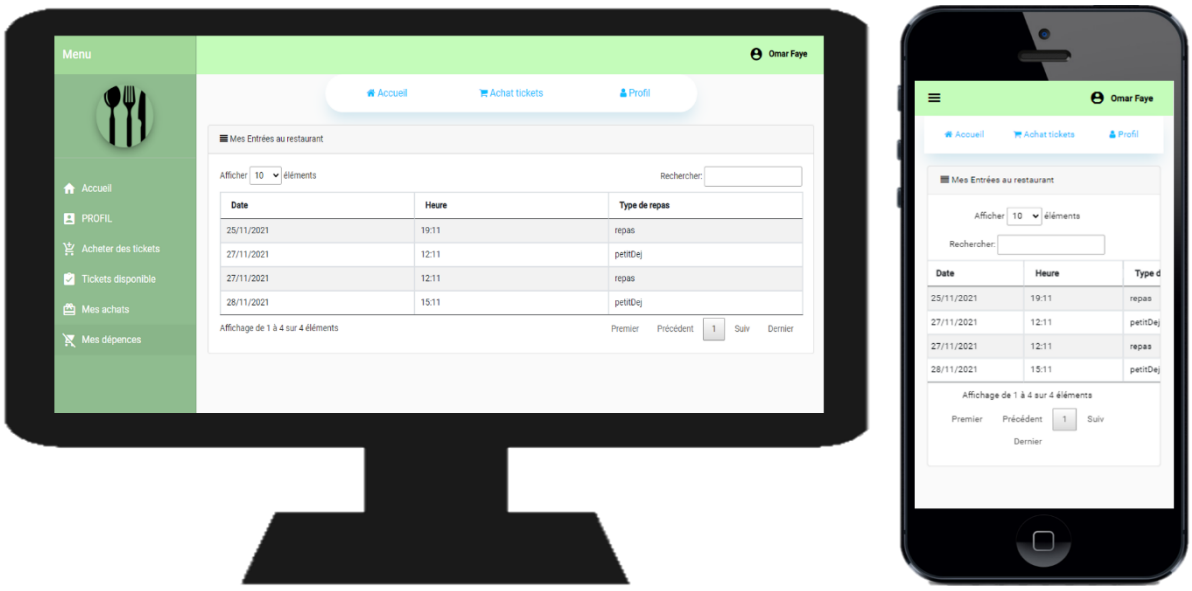


Figure 44 : : Historique de l'ensemble de ses entrées au restaurant universitaire

V.4 Les interfaces pour le vendeur de ticket

Le vendeur de ticket ou caissier bénéficie de trois pages principales dont une pour la vente de ticket, une page pour suivre ses ventes et de générer un rapport journalier.

V.4.1 L'interface pour vendre des tickets de restaurant

Cette interface permet au vendeur de ticket de pouvoir vendre des tickets aux étudiants. Pour faire une vente, il va devoir scanner le code QR de l'étudiant pour l'identifier et ensuite renseigner le nombre de petit déjeuner et de repas. L'application va faire le calcul et lui indiqué combien l'étudiant doit payer. Après validation de la vente, un reçu est automatiquement imprimé. La **figure 45** ci-dessous illustre cette page.

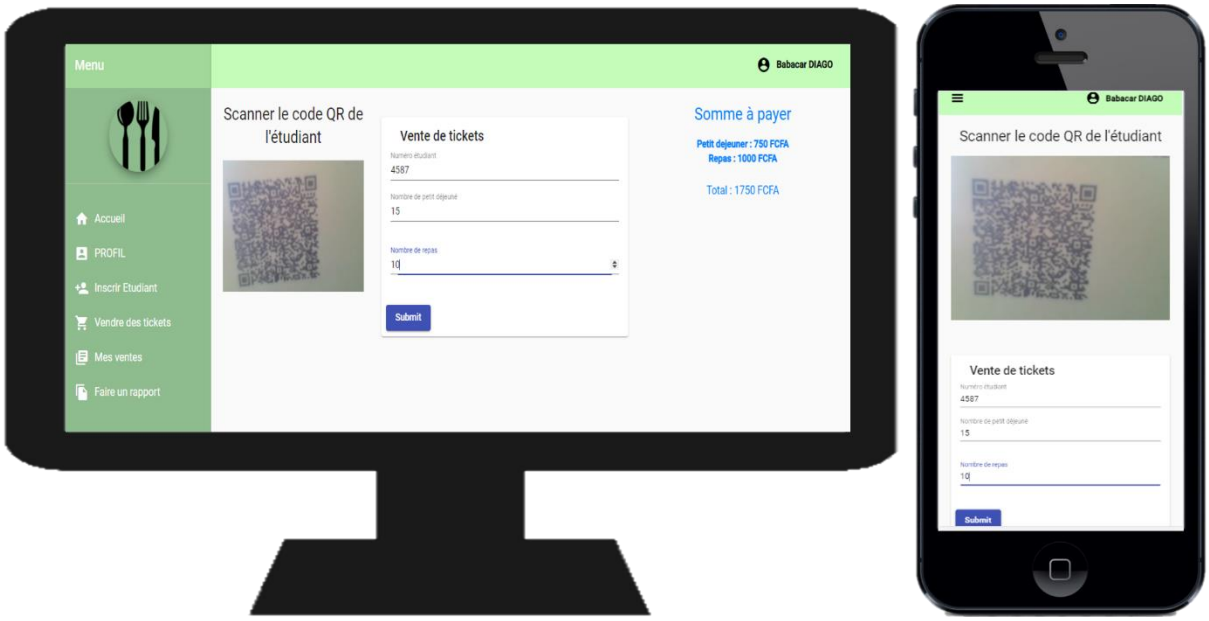


Figure 45 : Interface pour vendre des tickets de restaurant

V.4.2 L'interface pour suivre ses ventes en temps réel

Cette interface permet au vendeur de ticket de suivre le nombre de ticket vendu en temps réel et de voire l'historique de ses ventes. La **figure 46** ci-dessous illustre cette page.

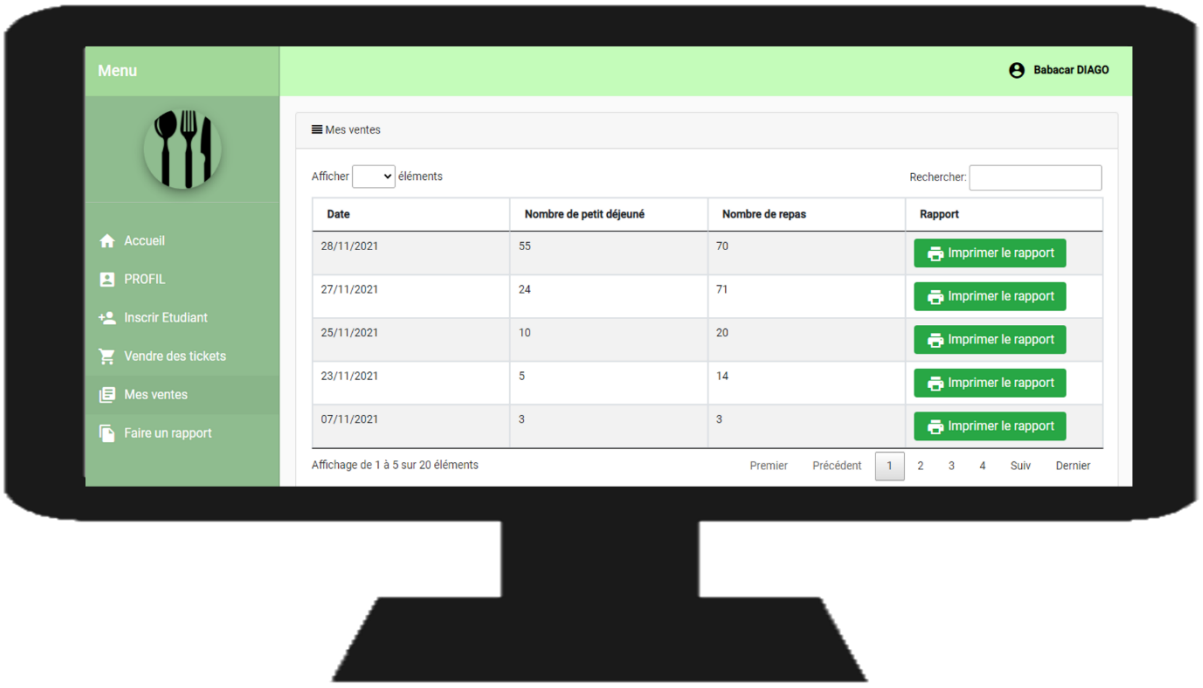


Figure 46 : Interface pour poursuivre les ventes de ticket en temps réel

V.4.3 L'interface pour générer un rapport journalier

Cette page permet au vendeur de ticket d'imprimer son rapport journalier. La **figure 47** ci-dessous montre cette page.

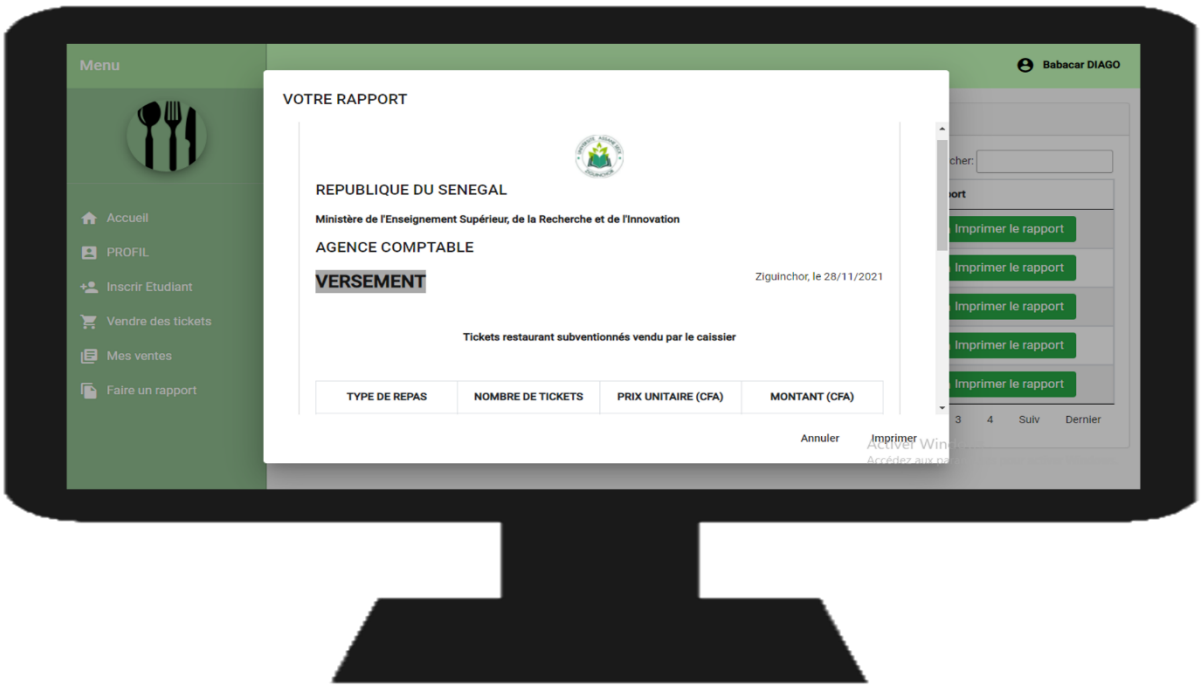


Figure 47 : Interface d'impression de rapport

V.5 L'interface pour le contrôle des accès au restaurant

Le contrôleur bénéficie principalement d'une interface lui permettant de faire le contrôle des accès des étudiants au restaurant. Cette interface permet aux contrôleurs de vérifier l'identité de l'étudiant et de vérifier si l'étudiant possède un ticket pour accéder au restaurant selon le type de repas. L'interface est dotée d'un lecteur QR qui scanne le QR de l'étudiant pour l'identifier et vérifier ses tickets. Si tout se passe bien, l'étudiant se verra débité d'un ticket pour accéder au restaurant. La **figure 48** ci-dessous permet d'illustrer cette interface.

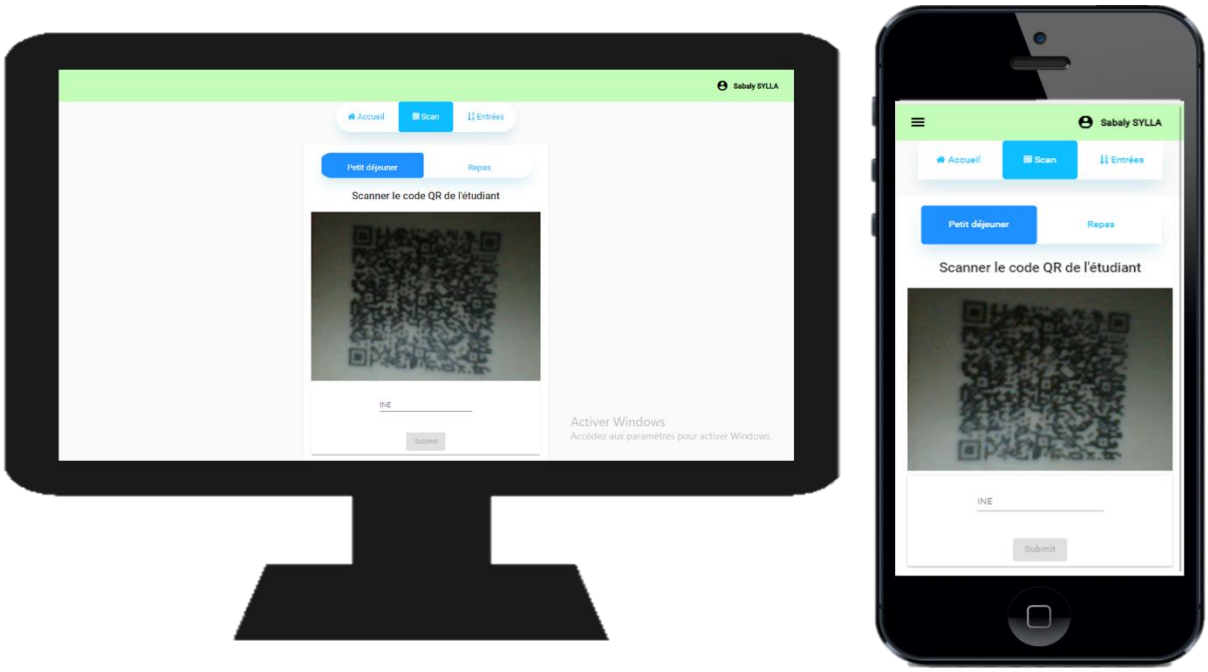


Figure 48 : Interface pour le contrôle des accès au restaurant

V.6 Les interfaces pour l'administrateur du système

L'administrateur du système peut ajouter, modifier, supprimer et attribué des rôles aux utilisateurs du système. C'est l'administrateur qui se charge d'inscrire l'ensemble des personnels administratifs qui doivent utiliser l'application. Nous avons utilisé **keycloak** pour la sécurisation du système et la gestion des utilisateurs.

Keycloak est un produit logiciel open source qui permet l'authentification unique (IdP) avec Identity Management et Access Management pour les applications et services modernes. Il permet aux développeurs de se concentrer sur les fonctionnalités métier en n'ayant pas à se soucier des aspects de sécurité de l'authentification.

J'ai développé aussi un microservice grasse à spring admin. Ce microservice permet à l'administrateur de surveiller l'ensemble des microservices déployé. Il lui fournit l'ensemble des instances de chaque microservice, leurs statuts qui est up ou down. Mais aussi des informations sur l'environnement où sont déployé les microservices. Ce microservice lui permet aussi d'avoir accès au journal des microservices.

V.6.1 L'interface pour la gestion des utilisateurs

A partir de cette interface, l'administrateur peut ajouter, modifier et supprimer un utilisateur. La **figure 49** ci-dessous permet d'illustrer cette interface.

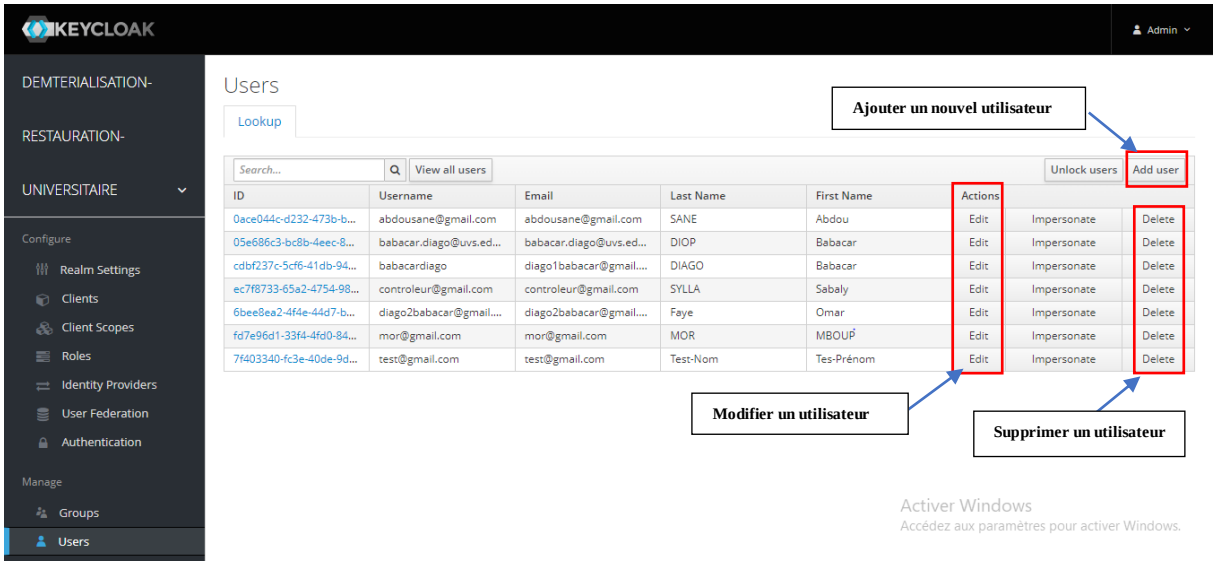


Figure 49 : Interface pour la gestion des utilisateurs

IV.6.2 L'interface pour la gestion des rôles des utilisateurs

L'administrateur est chargé d'attribuer aux utilisateurs des rôles ou droit d'accès. Chaque utilisateur a le droit d'accéder à certaines fonctionnalités de l'application selon son rôle. La **figure 50** ci-dessous montre en image la page de gestion des rôles des utilisateurs.

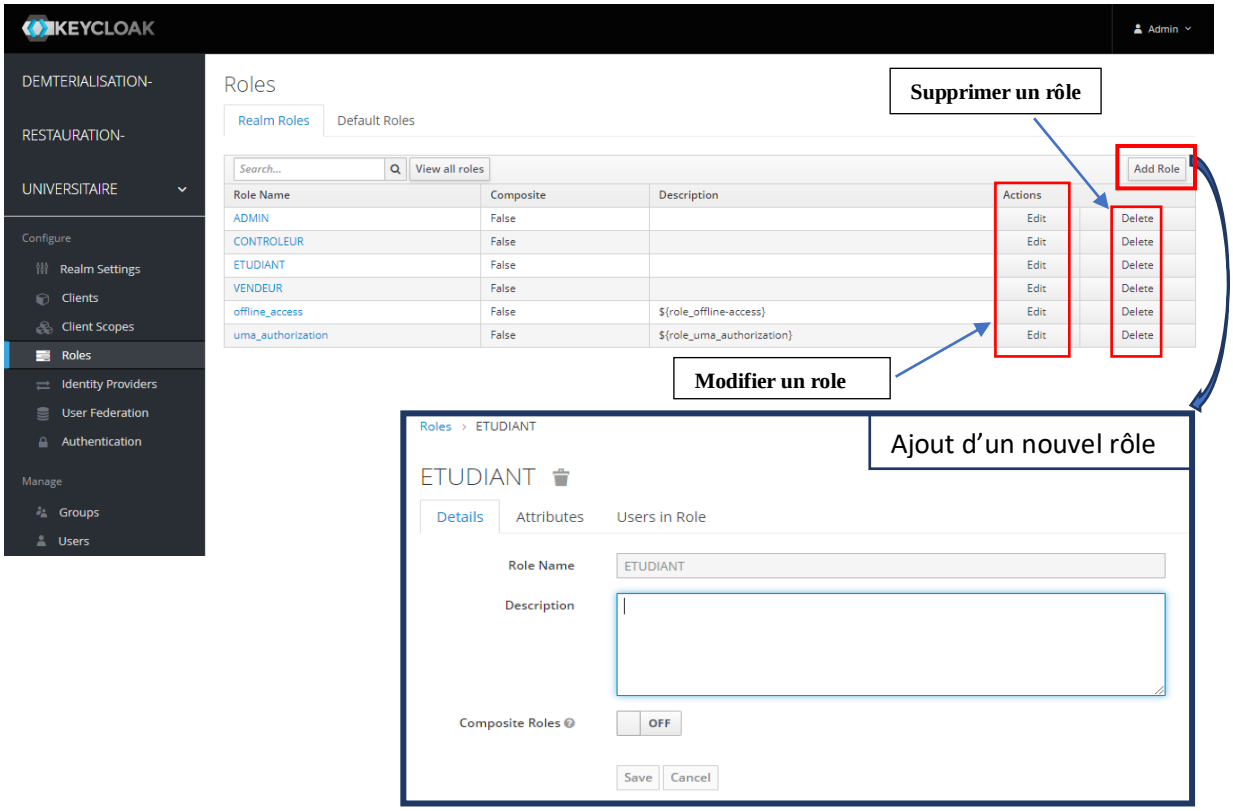


Figure 50 : Interface pour la gestion des rôles des utilisateurs

IV.6.3 L'interface pour le suivi de la disponibilité des microservices

L'administrateur a aussi la responsabilité de surveiller la disponibilité des différentes instances des microservices déployés. Les **figures 51 et 52** montrent les tableaux de bord qui nous renseigne sur l'état de chaque microservice.

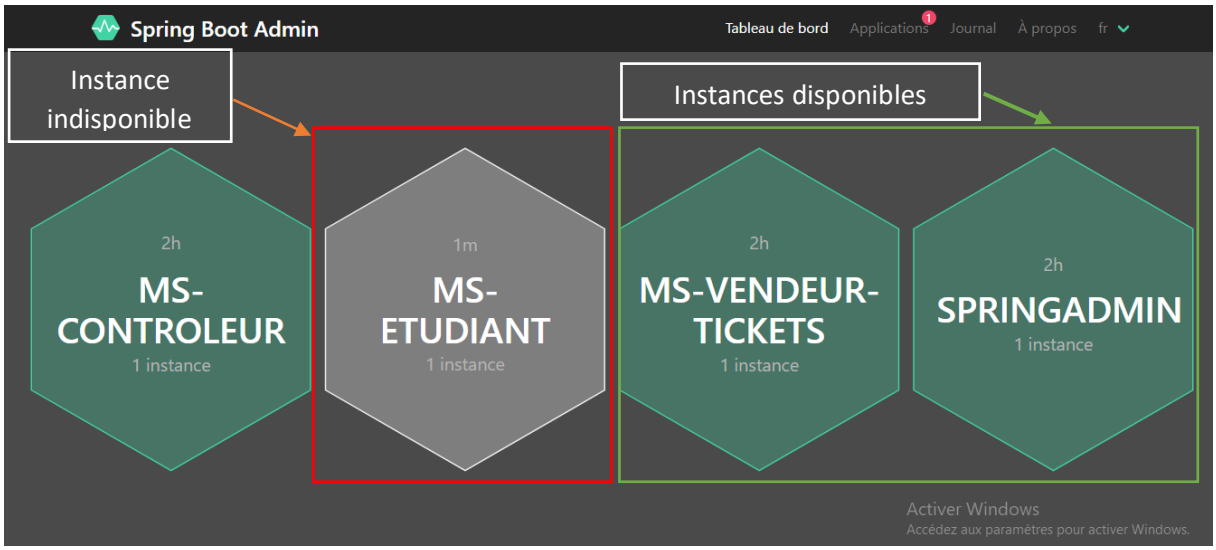


Figure 51 : tableau de bord montrant les microservices déployés

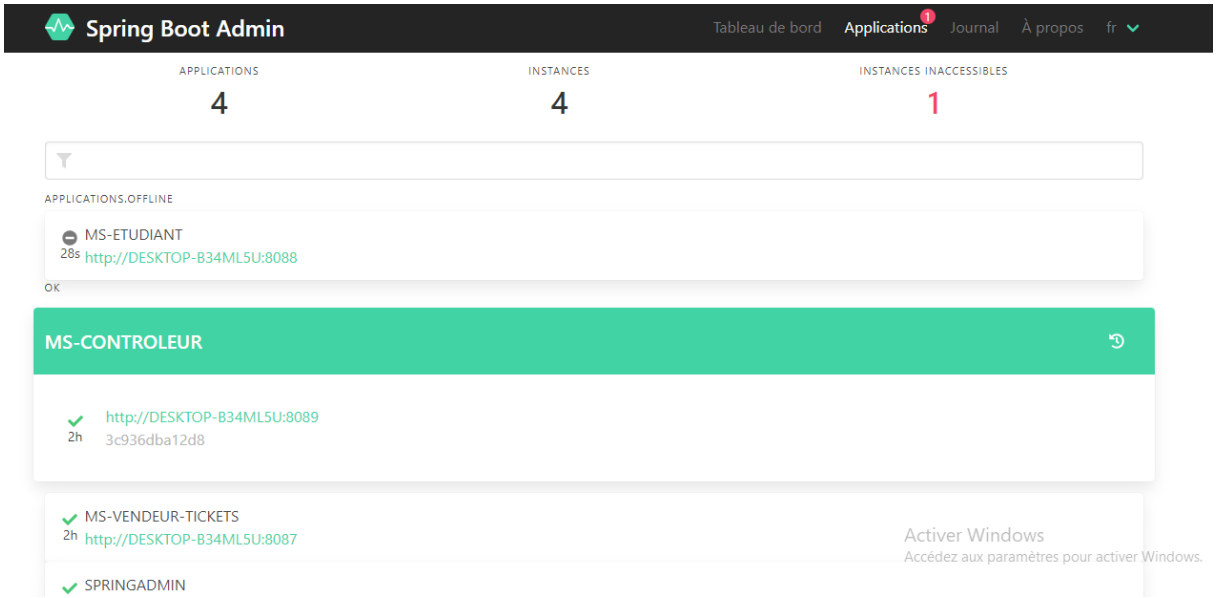


Figure 52 : Tableau de bord plus en détails

Les figures ci-dessous nous permettent de voir la disponibilité des microservice déployés et leurs ports. Ils nous permettent aussi de voir depuis quand le microservice est disponible ou indisponible.

IV.6.4 L'interface pour le suivi du journal des microservices

Le journal des microservice nous permet de les tracer. Il nous permet de savoir les différents états de chaque microservice et à quelle date et heure. Les **figures 53 et 54** illustrent les journaux d'évènements des microservices.

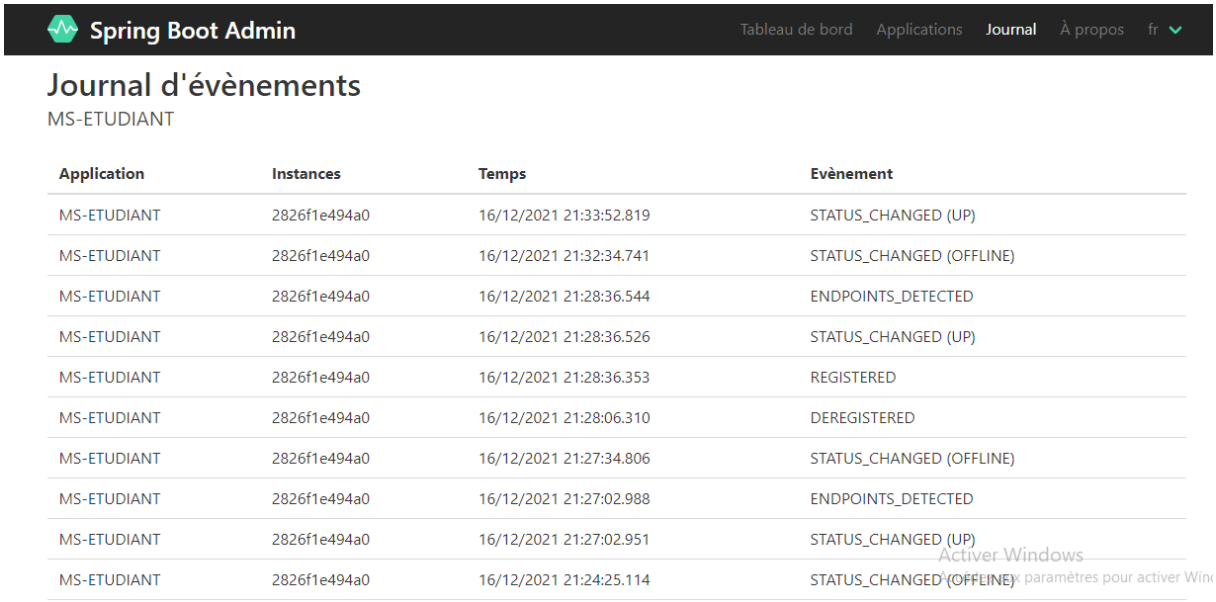


Figure 53 : Journal d'évènements du microservice Etudiant

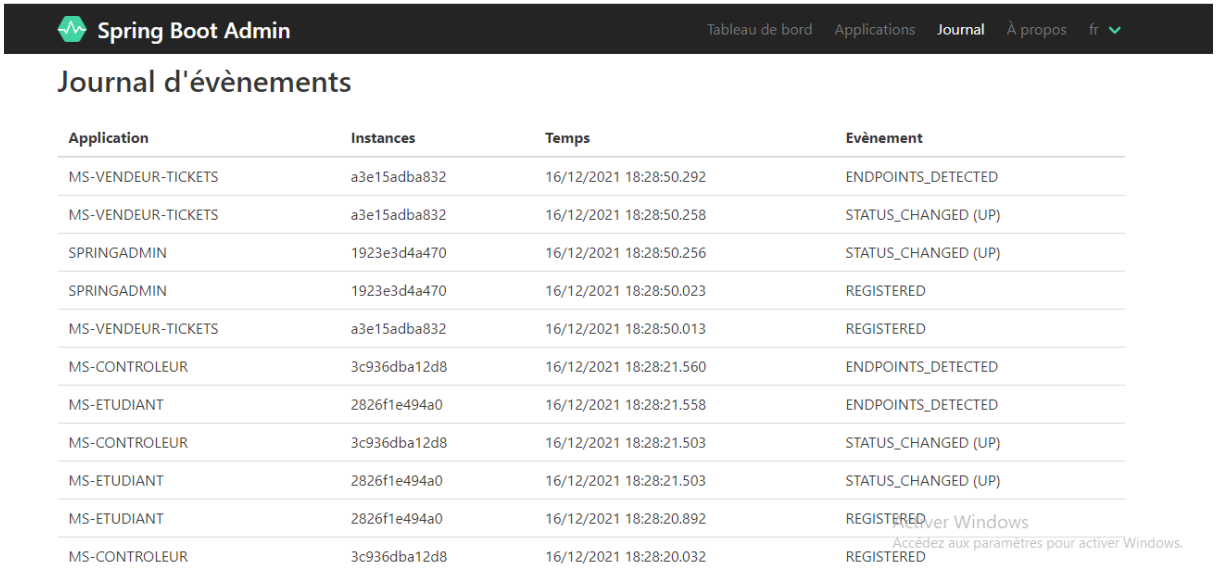


Figure 54 : Journal d'évènements des microservices

Conclusion

Dans ce chapitre, nous avons présenté les interfaces de l'application en spécifiant les utilisateurs qui vont en avoir accès. L'application est responsive.

La partie qui suit fait une conclusion générale de ce présent mémoire, ouvrant aussi des perspectives pour une futur amélioration de notre système.

CONCLUSION

En définitive l'objectif de ce mémoire était d'automatiser le système de la vente des tickets et de l'accès au restaurant universitaire. Pour cela, nous avons réalisé une application interactive basée sur les microservices permettant de gérer les différents traitements du système et de satisfaire les besoins des différents utilisateurs impliqués dans ce processus.

Nous avons commencé ce travail par la compréhension du contexte du sujet. Ensuite, nous avons réalisé une étude dans ce domaine, afin de pouvoir noter les manquements et les objectifs pour avoir un système satisfaisant. Après, nous sommes passé à l'étude conceptuelle de l'application selon une approche orientée objet tout en se basant sur le langage UML et le processus unifié 2TUP. Enfin, nous avons effectué l'implémentation de l'application.

Cette application consiste à dématérialiser les tickets de restauration et permettre aux étudiants de pouvoir acheté des tickets sans se déplacer grasse à la porte money Univ-Money. Chaque étudiant est identifié par un codeQR. Elle permettra aussi aux contrôleurs d'accès au restaurant universitaire d'avoir un système de contrôle d'accès qui leurs permettra d'identifier les étudiants grâce à un lecteur QR et de leurs débiter un ticket. Quant aux caissiers (vendeurs de tickets), l'application les permettra de vendre des tickets aux étudiant qui n'ont pas accès à leur compte ou qui n'ont pas de porte money.

Ce projet est venu innover le système de restauration universitaire car il facilite sa gestion et favorise la transparence. Il n'existe pas d'université sénégalaises qui l'implémente pour le moment.

En perspective, nous envisageons de déployer l'application afin d'élargir ce projet dans toutes les universités du Sénégal, mais aussi d'améliorer cette application en ajoutant d'autres fonctionnalités comme :

- Intégrer d'autres moyens de paiement en ligne comme Orange Money, Wave et Free Money
- le transfert de tickets entre étudiants ;
- une version mobile destinée aux étudiants afin de leur permettre d'installer l'application dans leurs téléphones ;
- installer des capteurs de présence pour permettre aux étudiants de voir en temps réel les files d'attente au niveau des RU.

- mettre en place un système décisionnel complet qui permettra aux décideurs de prendre les bonnes décisions pour une meilleur prise en charge des étudiants et des personnels.

Annexe 1 : Fiche d'approvisionnement

REPUBLIQUE DU SENEGAL
****
Ministère de l'Enseignement Supérieur,
de la Recherche et de l'Innovation
****
Centre Régional des Œuvres Universitaires Sociale de Ziguinchor
****
AGENCE COMPTABLE
****

ZIGUINCHOR, le 15/01/2021

APPROVISONNEMENT [redacted]

Les tickets restaurant subventionnés remis au caissier

. Déjeuner

NUMERO DE SERIE	QUANTITE	QUOTITE	Montant
930 001 - 940 000	10 000	100	1 000 000
940 001 - 950 000	10 000		1 000 000
TOTAL	**10 000**		**2 000 000**

PETIT DEJEUNER

NUMERO DE SERIE	QUANTITE	QUOTITE	Montant
330 001 - 335 000		50	250 000
335 001 - 340 000	5 000		250 000
TOTAL	**5 000**		**500 000**
MONTANT TOTAL			**2 500 000**

CAISSIER : [redacted]

CHEF BUREAU RECOUVREMENT

Annexe 2 : Fiche de versement

REPUBLIQUE DU SENEGAL
****
Ministère de l'Enseignement Supérieur,
de la Recherche et de l'Innovation
****

****
AGENCE COMPTABLE
****

ZIGUINCHOR, le 18/01/2021

[redacted]

VERSEMENT

Les tickets restaurant subventionnés remis au caissier

. Déjeuner

NUMERO DE SERIE	NOMBRE DE TICKETS	QUOTITE	Montant
900 001 - 910 000	**10 000**	100	**1 000 000**
910 001 - 920 000	**10 000**		**1 000 000**
920 001 - 925 000	**5 000**		**500 000**
TOTAL	**20 000**		**2 500 000**

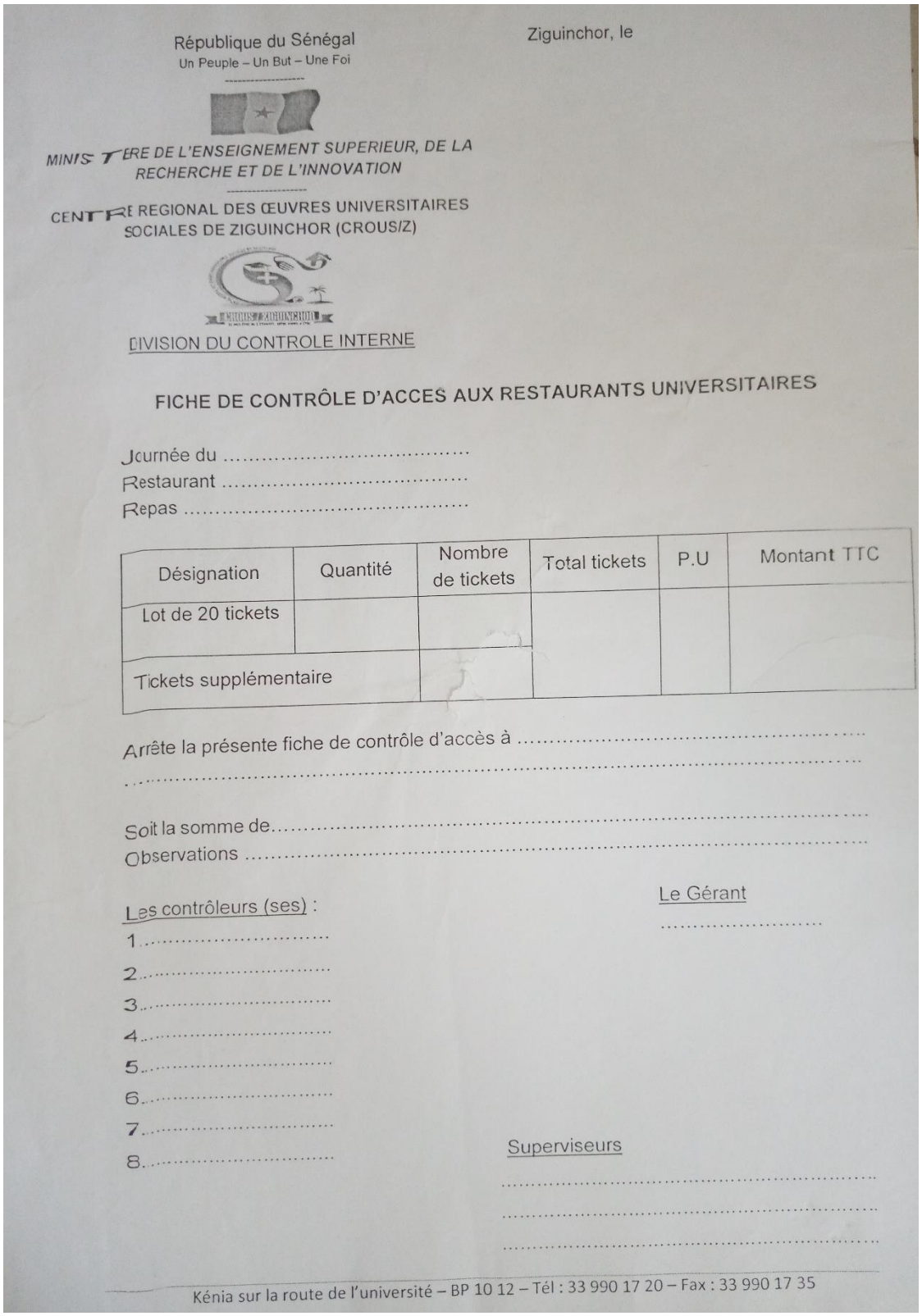
TOTAL VERSEMENT	2 500 000

CAISSIER : [redacted]

CHEF BUREAU RECOUVREMENT

LE CAISSIER PRINCIPAL

Annexe 3 : Fiche de contrôle d'accès aux restaurants universitaires



République du Sénégal
Un Peuple – Un But – Une Foi

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR, DE LA RECHERCHE ET DE L'INNOVATION

CENTRE REGIONAL DES ŒUVRES UNIVERSITAIRES SOCIALES DE ZIGUINCHOR (CROUS/Z)

DIVISION DU CONTROLE INTERNE

Ziguinchor, le

FICHE DE CONTRÔLE D'ACCES AUX RESTAURANTS UNIVERSITAIRES

Journée du .....
Restaurant .....
Repas .....

Désignation	Quantité	Nombre de tickets	Total tickets	P.U	Montant TTC
Lot de 20 tickets					
Tickets supplémentaire					

Arrête la présente fiche de contrôle d'accès à .....
.....

Soit la somme de.....
Observations .....

Les contrôleurs (ses) :
1.....
2.....
3.....
4.....
5.....
6.....
7.....
8.....

Le Gérant
.....

Superviseurs
.....
.....
.....

Kénia sur la route de l'université – BP 10 12 – Tél : 33 990 17 20 – Fax : 33 990 17 35

BIBLIOGRAPHIE ET WEBOGRAPHIE

Monsieur Mor MBOUP. « Univ-money, un porte-monnaie électronique pour la gestion des transactions étudiantes au sein de l'université », Mémoire de Master 2 Génie Logiciel soutenu le 08/12/2021 à l'UASZ

[1] **Sophnouill.** developpez.com, Publié le 4 mai 2004 - Mis à jour le 19 juin 2020, consulté le 19/09/2021, Disponible sur : https://sabricole.developpez.com/uml/tutoriel/unifiedProcess/

[2] **SCRIBD.** https://fr.scribd.com, Transféré par badrou Date du transfert le 28/02/2011, consulté le 19/09/2021, Disponible sur :

https://fr.scribd.com/doc/49697489/Processus-de-Developpement-Y-Processus-2TUP

[3] **Wikipedia.** https://fr.wikipedia.org, dernière modification le 30/10/ 2021, Consulté le 23/09/2021, Disponible sur : https://fr.wikipedia.org/wiki/Processus_unifi%C3%A9#Variantes_du_processus_unifi%C3%A9

[4] **Wikipedia.** https://fr.wikipedia.org, dernière modification le 09/01/2021, Consulté le 20/09/2021, Disponible sur : https://fr.wikipedia.org/wiki/Two_Tracks_Unified_Process

www.e-bancel.com, Consulté le 23/09/2021 https://www.e-bancel.com/Processus_2TUP.php

[5] : **Spring**, https://spring.io, Consulté le 10/11/2021, Disponible sur : https://spring.io/microservices

[Figure 1] https://www.shutterstock.com, Consulté le 10/11/2021, Disponible sur : https://www.shutterstock.com/fr/search/microservices

[Figure2] https://docs.oracle.com, Consulté le 10/11/2021, Disponible sur : https://docs.oracle.com/fr/solutions/learn-architect-microservice/index.html#GUID-1A9ECC2B-F7E6-430F-8EDA-911712467953

[6] : **Oracle.** https://docs.oracle.com, Consulté le10/11/2021, Disponible sur : https://docs.oracle.com/fr/solutions/learn-architect-microservice/index.html#GUID-5830BC1D-CC15-4A3A-B9ED-8A8D84067D7A

[7] : **Lucidchart.** https://www.lucidchart.com, Consulté le 13/11/2021, Disponible sur : https://www.lucidchart.com/pages/fr/diagramme-de-composants-uml

[8] : **Lucidchart.** https://www.lucidchart.com, Consulté le 13/11/2021, Disponible sur : https://www.lucidchart.com/pages/fr/diagramme-package-uml

[9] : **edrawsoft**. https://www.edrawsoft.com, Consulté le 20/11/2021, Disponible sur : https://www.edrawsoft.com/fr/uml-deployment-diagramsolutions.html?gclid=CjwKCAiA1aiMBhAUEiwACw25MQX3U00UV2BdIFCER6qmicwSB5_hbi4---z3mFywm7C8Ht03pGddsRoC8qQQAvD_BwE

[10] : **Docker doc.** https://docs.docker.com, Consulté le 25/11/2021, Disponible sur : https://docs.docker.com/engine/

http://remy-manu.no-ip.biz, Consulté le 27/11/2021, Disponible sur : http://remy-manu.no-ip.biz/UML/Cours/coursUML10.pdf

[11] : **Orange.** https://moodle.campusafrica.gos.orange.com, Publié en Septembre 2020, consulté le 27/11/2021, Disponible sur : https://moodle.campusafrica.gos.orange.com/pluginfile.php/2890/mod_resource/content/1/FormationSpringBoot-4-SpringData.pdf

[12] : **Open classrooms.** https://openclassrooms.com/fr, Mis à jour le 27/09/2021, consulté 08/12/2021, Disponible sur : https://openclassrooms.com/fr/courses/6900101-creez-une-application-java-avec-spring-boot/7078015-creez-un-controleur-rest-pour-gerer-vos-donnees

[13] : **John Thompson.** https://dzone.com, Publié le 17/06/2021, consulté le 10/12/2021, Disponible sur : https://dzone.com/articles/spring-boot-restful-api-documentation-with-swagger