

Ecole Marocaines des Sciences de l'Ingénieur  
Département Génie Informatique

---

Rapport de Projet de POO  
Gestion d'une Cafétéria

---

Durant la période : du 01/03/2023 a 03/04/2024

Encadrant

Madame Hazman  
Madame Sniba  
Monsieur Safsouf

Etudiants

Ouhmida Soulaïmane  
Ettakaddoumi Hamza  
Targui Hajar  
Kelladi Fatima ezzahra

## Dedicase

À mes collègues et partenaires de travail, pour leur collaboration et leur engagement tout au long de ce projet. Vos idées, votre soutien et votre camaraderie ont été essentiels pour surmonter les défis rencontrés en cours de route.

À ma famille, pour leur soutien inconditionnel et leur compréhension pendant les longues heures de travail et les moments de stress. Votre amour et votre soutien sont ma source d'inspiration quotidienne.

À toutes les personnes impliquées dans ce projet, qu'ils soient mentionnés ici ou non, je vous suis reconnaissant pour votre contribution et votre dévouement. Ce projet n'aurait pas été possible sans votre collaboration.

À ceux qui croient en l'importance de l'innovation et du progrès, cette dédicace est pour vous. Votre soutien à la recherche de nouvelles idées et de solutions créatives est ce qui alimente notre motivation à poursuivre.

## Remerciement

Je tiens à exprimer notre profonde gratitude envers toutes les personnes qui ont contribué à la réalisation de ce projet. Tout d'abord, nous souhaitons remercier les Profs. **SAFSOUF**, **SNIBA** et **HAZMAN** pour leur direction et leurs conseils précieux tout au long du processus. Leur vision et leur expertise ont été des éléments essentiels pour atteindre nos objectifs.

Nos remerciements vont également à tous les membres de notre équipe pour leur dévouement, leur travail acharné et leur collaboration exceptionnelle. Chacun de vous a apporté une contribution unique qui a enrichi notre projet et a permis d'atteindre les résultats que nous avons obtenus.

Ce projet a été une expérience enrichissante grâce à la contribution de chacun. Nous sommes reconnaissants d'avoir eu l'opportunité de travailler avec une équipe aussi talentueuse et dévouée. Merci à tous pour votre engagement et votre travail acharné.

## Abstract

L'objectif de ce projet de gestion de cafétéria est de développer un système informatisé permettant de rationaliser et d'optimiser les opérations quotidiennes d'une cafétéria. Le système comprendra des fonctionnalités telles que la gestion des stocks, la prise de commandes, le suivi des ventes, la gestion des employés et des horaires, ainsi que la génération de rapports pour faciliter la prise de décision et améliorer l'efficacité opérationnelle. En utilisant cette solution, la cafétéria pourra réduire les coûts, minimiser les pertes liées aux stocks, améliorer la satisfaction des clients et optimiser la gestion des ressources humaines.

## Résumé

Le projet de gestion de cafétéria consiste en la conception et le développement d'un système informatisé complet destiné à rationaliser toutes les activités opérationnelles d'une cafétéria. Ce système sera doté de fonctionnalités détaillées, telles que la gestion détaillée des stocks incluant la gestion des fournisseurs, la réception des marchandises et le suivi des niveaux de stock en temps réel. Il permettra également la prise de commandes efficace, en offrant aux clients la possibilité de passer des commandes en ligne ou sur place, tout en garantissant une gestion précise des paiements et des transactions.

De plus, le système inclura un module de gestion des employés permettant de gérer les horaires de travail, les affectations de tâches et le suivi des performances individuelles. Les fonctionnalités de reporting seront également intégrées pour générer des rapports détaillés sur les ventes, les tendances de consommation, les niveaux de stock et les performances du personnel. Ces rapports seront essentiels pour prendre des décisions stratégiques et améliorer continuellement les opérations de la cafétéria.

## Glossaire

Mot	Signification
CRUD	CRUD est un acronyme issu du monde de la programmation informatique et fait référence aux quatre fonctions considérées comme nécessaires pour mettre en œuvre une application de stockage persistant : créer, lire, mettre à jour et supprimer.
EMSI	École Marocaine des Sciences de l'Ingénieur
TIC	Technologies de l'information et de la communication
REST	ne interface de programmation d'application (API ou API web) qui respecte les contraintes du style d'architecture REST et permet d'interagir avec les services web RESTful.
API	Application programming interface ou interface de programmation d'application
HTTP	L'HyperText Transfer Protocol
DOM	Document Object Model, C'est une représentation hiérarchique d'une page web qui permet aux programmes JavaScript et aux langages de script d'accéder et de manipuler le contenu, la structure et le style d'une page web.

## Liste des figures

Figure 1: le logo de l'EMSI.....	14
Figure 2: Les acteurs et relation d'héritage.....	17
Figure 3: Diagramme général de cas d'utilisation.....	18
Figure 4: Cas d'utilisation "Client".....	19
Figure 6: Cas d'utilisation "Administrateur".....	20
Figure 7: Diagramme de classes.....	22
Figure 37: : Interactions entre le modèle, la vue et le contrôleur.....	61
Figure 38: Page d'accueil du postman. ....	62
Figure 39: Comment fonctionne Postman?.....	62
Figure 40: Requet http "GET".....	63
Figure 41: Requet http "POST".....	64
Figure 42: Requet http "PATCH".....	64
Figure 43: Requet http "DELETE".....	65
Figure 44: Comment fonctionne le commande ? .....	66
Figure 46: la page de paiement.....	66
Figure 48: La commande sur Stripe.....	67

## Liste des tableaux

Table 1: Description Textuelle « Commander les produits » .....	19
Table 2: Description Textuelle « Consulter les recommandations » .....	20
Table 6: Description Textuelle « Gestion des utilisateurs ».....	20
Table 7: Description Textuelle « Gestion des commandes ».....	21

## Table of Contents

Dedicase .....	2
Remerciement .....	3
Abstract .....	4
Résumé .....	5
Glossaire .....	6
Liste des figures .....	7
Liste des tableaux .....	7
Introduction Générale .....	11
Sujet Amené : Contexte et Cadre de l'Étude .....	11
Sujet Posé : Problématique, Objectifs et Démarche .....	11
Sujet Divisé : Plan de Rédaction .....	11
<b>Chapitre1: L'organisme d'accueil</b>	
Introduction .....	14
Conclusion .....	15
<b>Chapitre 2: Analyse et Conception</b>	
Introduction .....	17
I. Diagramme de cas d'utilisation .....	17
1. Identification des acteurs du système .....	17
2. Diagramme général de cas d'utilisation .....	18
3. Description textuelle des cas généraux d'utilisation .....	19
a. Module Client .....	19
b. Module Administrateur .....	20
II. Diagramme de classes .....	21
III. Diagramme de sequences .....	23
Conclusion .....	26
<b>Chapitre 3: Base de données</b>	
Introduction .....	28
C'est quoi base de données? .....	28
Choix du système de gestion de la base de données .....	28



Implémentation de base de données .....	29
• Table Users.....	29
• Table Categories.....	29
• Table Dishes.....	29
• Table Favorites .....	30
• Table Tables .....	30
• Table Reservations.....	30
• Table Orders .....	31
• Table Paiements .....	31

#### Chapitre 4: Outils de travaux

Introduction .....	33
React.....	33
React Router DOM.....	33
Axios .....	34
Redux .....	34
Postman .....	34
Stripe.....	35
Express.....	35
Microsoft SQL Server.....	35
GitHub .....	36
Visual Studio Code .....	36
Conclusion .....	36

#### Chapitre 5 : Mise en œuvre et réalisation

Introduction .....	38
I. Server (Back End) .....	38
a. Présentation du framework express js.....	38
Pourquoi Express js?.....	38
Rôle de Express .....	38
L'application Express.js.....	38
a. Mise en place du backend de conception-MVC (voir annexe).....	40
b. Rôle de Express .....	41

c. Fonctionnement du serveur .....	41
d. Les classes .....	42
Dishe .....	42
Categorie .....	44
Favorite.....	46
Table.....	49
Users .....	52
Reservation .....	54
Order.....	56
Conclusion .....	58
Conclusion et perspective .....	59
Annexe.....	60
I. L'architecture MVC et l'utilisation des Frameworks .....	60
a. Les Frameworks .....	60
b. L'architecture MVC .....	60
II. Gestion des produits avec Postman.....	62
1. La page principale de Postman.....	62
2. Comment fonctionne Postman?.....	62
HTTP Requests .....	63
Responses.....	63
III. Comment la Commande travailler sous couverture.....	66
a. Valider le paiement .....	66
b. Consulter la commande .....	67
Bibliographie.....	68

# Introduction Générale

## Sujet Amené : Contexte et Cadre de l'Étude

Dans un monde où les cafétérias occupent une place centrale dans la vie quotidienne, répondant aux besoins alimentaires et sociaux de divers publics, leur gestion représente un défi complexe. Cette étude vise à explorer les défis spécifiques rencontrés par les gestionnaires de cafétérias et à identifier des opportunités pour améliorer leur performance. En analysant le cadre réglementaire, les tendances du marché et les attentes des consommateurs, elle fournira des recommandations pour renforcer l'efficacité opérationnelle, la rentabilité et la satisfaction client dans ce secteur en constante évolution.

## Sujet Posé : Problématique, Objectifs et Démarche

**Problématique :** La gestion des cafétérias est confrontée à une multitude de défis, allant de la fluctuation des préférences des consommateurs à la gestion efficace des ressources et des coûts. Comment les gestionnaires de cafétérias peuvent-ils surmonter ces défis pour améliorer l'efficacité opérationnelle, la satisfaction client et la rentabilité de leur établissement ?

L'objectif est d'identifier les principaux défis rencontrés par les gestionnaires de cafétérias dans leur quotidien. Analyser les attentes et les préférences des consommateurs en matière de restauration dans les cafétérias. Proposer des stratégies et des recommandations pour optimiser la gestion des stocks, la planification des menus et la satisfaction client. Évaluer l'impact de ces stratégies sur l'efficacité opérationnelle et la rentabilité des cafétérias.

Il convient de noter que ce rapport ne vise pas à présenter les résultats du projet, mais plutôt à décrire le processus de développement et les choix méthodologiques effectués.

## Sujet Divisé : Plan de Rédaction

**Analyse des Besoins Utilisateurs :** Cette section explorera les attentes et les besoins des utilisateurs en matière de gestion de cafétéria. Nous examinerons les exigences spécifiques des gestionnaires de cafétéria en termes de gestion des stocks, de prise de commandes, de gestion du personnel et de satisfaction client. De plus, nous étudierons

les attentes des clients finaux en ce qui concerne l'expérience de commande, les options de menu, les délais de service et la facilité d'utilisation de la plateforme en ligne.

**Conception de l'Architecture :** Nous détaillerons ici l'architecture logicielle conçue pour répondre aux besoins identifiés dans la section précédente. Cela inclura la conception d'une interface conviviale pour les gestionnaires de cafétéria, permettant une gestion efficace des stocks et des commandes. Nous aborderons également la conception d'une interface utilisateur intuitive pour les clients finaux, leur permettant de passer des commandes de manière facile et rapide.

**Développement et Intégration :** Cette section traitera du processus de développement des fonctionnalités de la plateforme de gestion de cafétéria. Nous décrirons les différentes étapes de développement, y compris la programmation, les tests unitaires et l'intégration des différents modules pour assurer un fonctionnement harmonieux de l'ensemble du système.

**Tests et Évaluation :** Nous présenterons les tests réalisés pour évaluer la convivialité de la plateforme de gestion de cafétéria. Cela inclura des tests d'acceptation utilisateur pour vérifier si le système répond aux besoins spécifiques des gestionnaires de cafétéria et des clients finaux. Nous évaluerons également la fiabilité et la robustesse du système en effectuant des tests de charge et de performance.

**Conclusion et Perspectives :** Enfin, nous conclurons en résumant les résultats obtenus à partir de l'analyse des besoins utilisateurs, de la conception, du développement et des tests du système de gestion de cafétéria. Nous discuterons également des perspectives d'amélioration futures, notamment l'intégration de nouvelles fonctionnalités, l'optimisation des performances et l'adaptation aux évolutions technologiques et aux besoins changeants du marché de la restauration.

---

# CHAPITRE1: L'ORGANISME D'ACCUEIL

---

## Introduction



Figure 1: le logo de l'EMSI

L'École Marocaine des Sciences de l'Ingénieur (EMSI) est une institution d'enseignement supérieur située au Maroc, reconnue pour son excellence académique et son engagement envers la formation d'ingénieurs compétents et qualifiés. Au sein de l'EMSI, le Département de Génie Informatique se distingue par son programme académique complet et sa formation axée sur les technologies de l'information et de la communication (TIC).

Le Département de Génie Informatique de l'EMSI offre un cursus d'études approfondi couvrant les principaux domaines de l'informatique, tels que la programmation, les systèmes d'information, les réseaux informatiques, la sécurité informatique, l'intelligence artificielle, le génie logiciel, et bien plus encore. Les étudiants sont exposés à des cours théoriques de pointe ainsi qu'à des travaux pratiques concrets, leur permettant d'acquérir des compétences techniques solides et une compréhension approfondie des principes fondamentaux de l'informatique.

En plus de sa rigueur académique, le Département de Génie Informatique de l'EMSI se distingue par son engagement envers l'innovation et la recherche. Les étudiants sont encouragés à participer à des projets de recherche, des concours technologiques et des événements industriels pour développer leurs compétences et explorer de nouvelles tendances dans le domaine de l'informatique.

Les diplômés du Département de Génie Informatique de l'EMSI sont hautement prisés sur le marché du travail, tant au niveau national qu'international. Leur formation polyvalente et leur expérience pratique leur permettent de s'adapter rapidement aux exigences de l'industrie informatique et de contribuer de manière significative à des projets variés dans divers secteurs, tels que les technologies de l'information, les télécommunications, la finance, la santé, et bien d'autres encore.

En résumé, le Département de Génie Informatique de l'EMSI offre aux étudiants une formation de qualité, alignée sur les besoins du marché et axée sur l'excellence académique, les préparant ainsi à une carrière réussie dans le domaine passionnant et dynamique de l'informatique.

## Conclusion

En conclusion, le Département de Génie Informatique de l'EMSI se positionne comme un pilier de l'excellence académique et de l'innovation dans le domaine de l'informatique au Maroc. Son programme complet et équilibré, alliant théorie et pratique, prépare les étudiants à relever avec succès les défis de l'industrie informatique moderne. Avec un engagement ferme envers la recherche et l'innovation, ainsi qu'une forte réputation d'employabilité, l'EMSI offre une voie solide pour ceux qui aspirent à une carrière réussie et enrichissante dans le domaine dynamique des technologies de l'information et de la communication.

---

# CHAPITRE 2: ANALYSE ET CONCEPTION

---



## Introduction

Ce chapitre donnera une vision de conception à travers des diagrammes de cas d'utilisation, des diagrammes de séquences et des diagrammes de classes.

### I. Diagramme de cas d'utilisation

#### 1. Identification des acteurs du système

Les types d'acteurs qui participent au système sont les suivants :

- **Client** : un utilisateur final de l'application. Il peut consulter les plats, les ajouter aux favoris et au panier et les commander. Il possède un login et un mot de passe et accède à un nombre d'écrans et vues de l'application qui sont relatifs à sa responsabilité.
- **Administrateur** : Un utilisateur administrateur avec un niveau d'habilitation qui lui permet:
  - Gestion des utilisateurs.
  - Gestion des catégories.
  - Gestion des plats.
  - Gestion des commandes.

La figure ci-dessous représente les relations d'héritage entre ces acteurs :

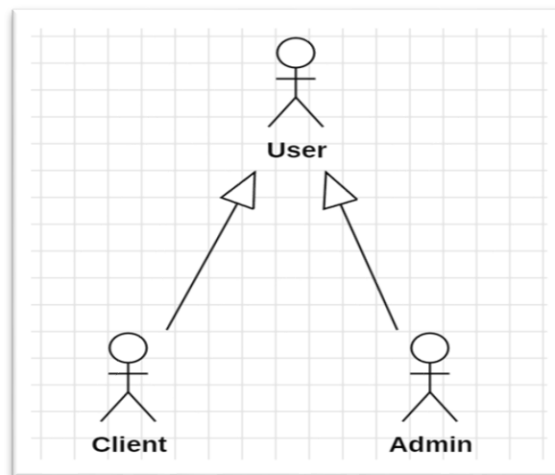


Figure 2: Les acteurs et relation d'héritage.

## 2. Diagramme général de cas d'utilisation

Le diagramme de cas d'utilisation ci-dessous représente l'ensemble des cas d'utilisations relative à l'application. Le but de ce diagramme est d'avoir une vision globale sur CLOE :

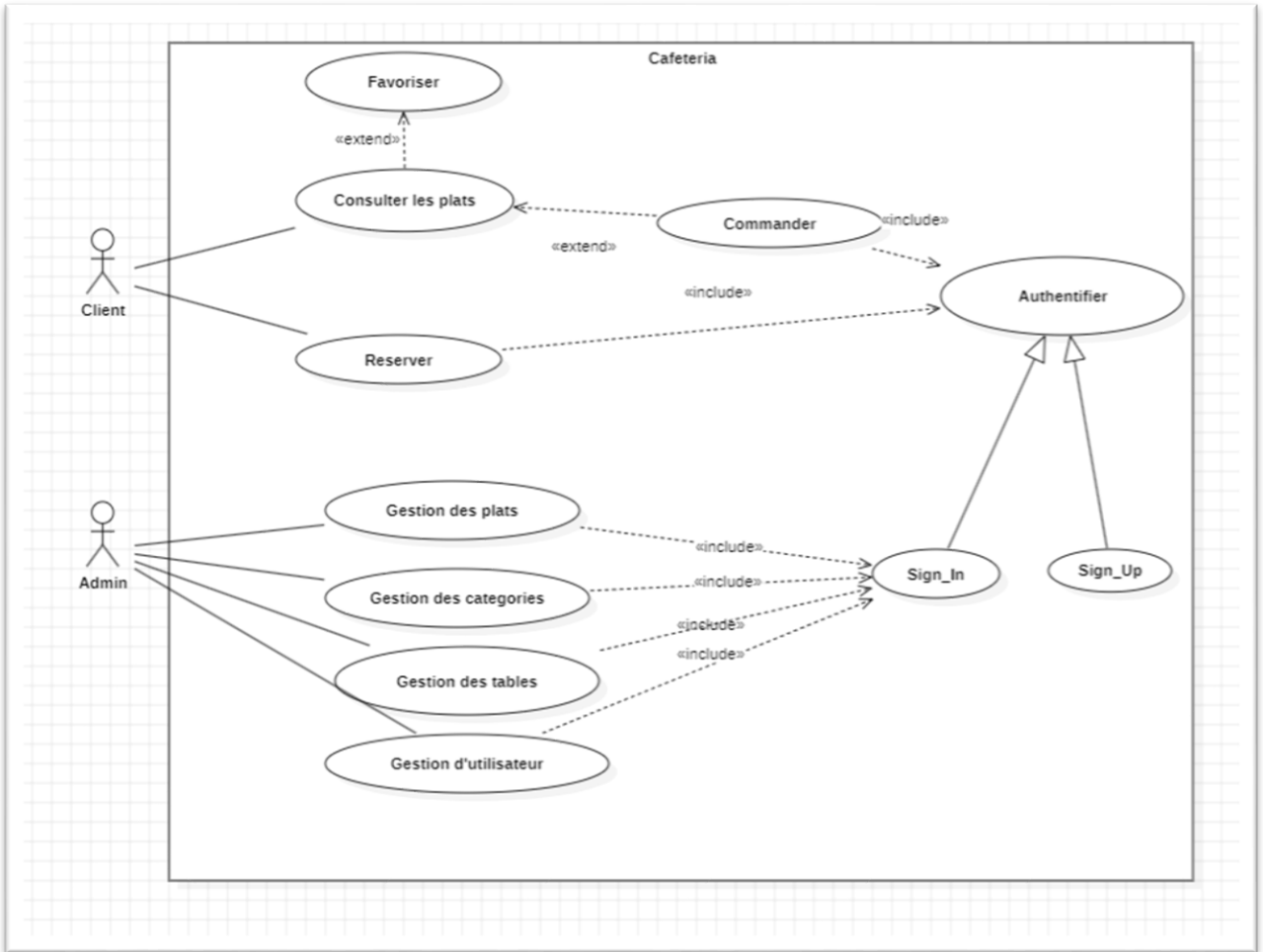


Figure 3: Diagramme général de cas d'utilisation

### 3. Description textuelle des cas généraux d'utilisation

#### a. Module Client

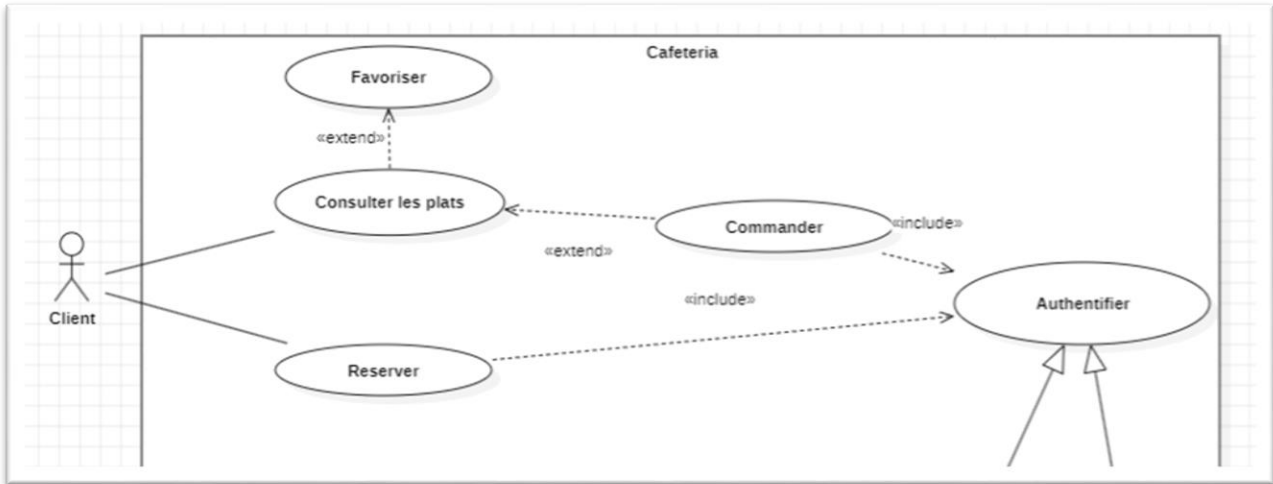


Figure 4: Cas d'utilisation "Client"

#### Description Textuelle:

On prend quelques exemples des cas d'utilisation.

Titre de scénario	Consulter les plats
Acteur	Client
Description	Ce cas d'utilisation permet au client de créer un compte, de consulter la liste des plats avec la possibilité de les ajouter à sa liste de favoris, puis de passer une commande.
Précondition	Authentification

Table 1: Description Textuelle « Commander les produits »

Titre de scénario	Reservation
Acteur	Client
Description	Ce cas d'utilisation permet au client de faire une reservation.
Précondition	Authentification

Table 2: Description Textuelle « Consulter les recommandations »

### ***b. Module Administrateur***

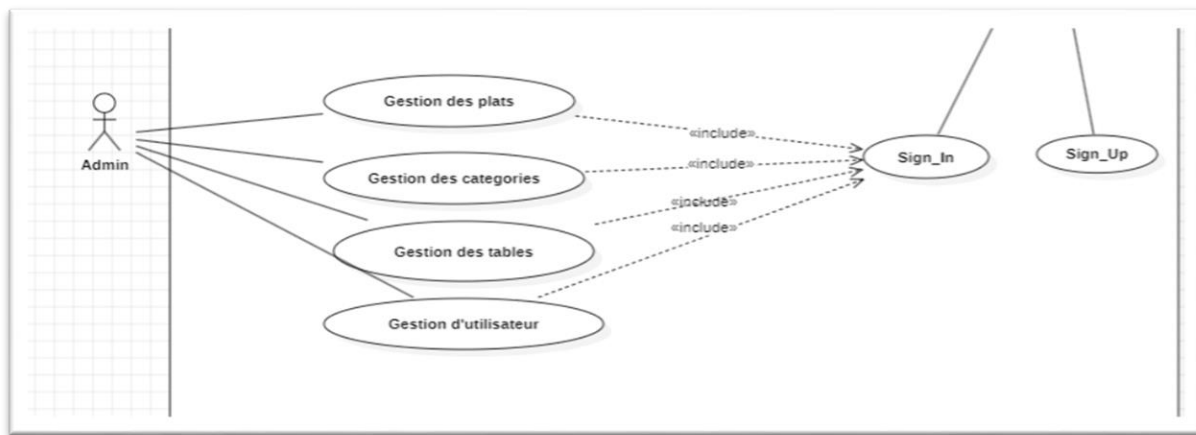


Figure 5: Cas d'utilisation "Administrateur"

### **Description Textuelle:**

On prend quelques exemples des cas d'utilisation.

Titre de scénario	Gestion des utilisateurs
Acteur	Administrateur
Description	Ce cas d'utilisation permet à l'administrateur de gérer les utilisateurs (clients, admins), et de gérer les commandes et les reservations.
Précondition	Authentification

Table 3: Description Textuelle « Gestion des utilisateurs »

Titre de scénario	Gestion des plats
Acteur	Administrateur
Description	. Ce cas d'utilisation permet au vendeur de gérer les plats (CRUD)
Précondition	Authentification

Table 4: Description Textuelle « Gestion des commandes »

## II. Diagramme de classes

Avant de commencer à concevoir le système d'information, et donc la base de données, il faut tout d'abord passer par la phase de la modélisation du schéma relationnel qui va implémenter la base de données.

Le schéma relationnel est réalisé après l'étude du système cultural, en respectant la méthode Merise ; qui est la référence pour la conception des bases de données.

Elle permet d'organiser les données du système d'information en suivant des règles et des normes. Cela permet de comprendre rapidement comment est structurée la base de données. Cette méthode a aidé à mettre en évidence les différentes entités et relations caractérisant le système cultural.

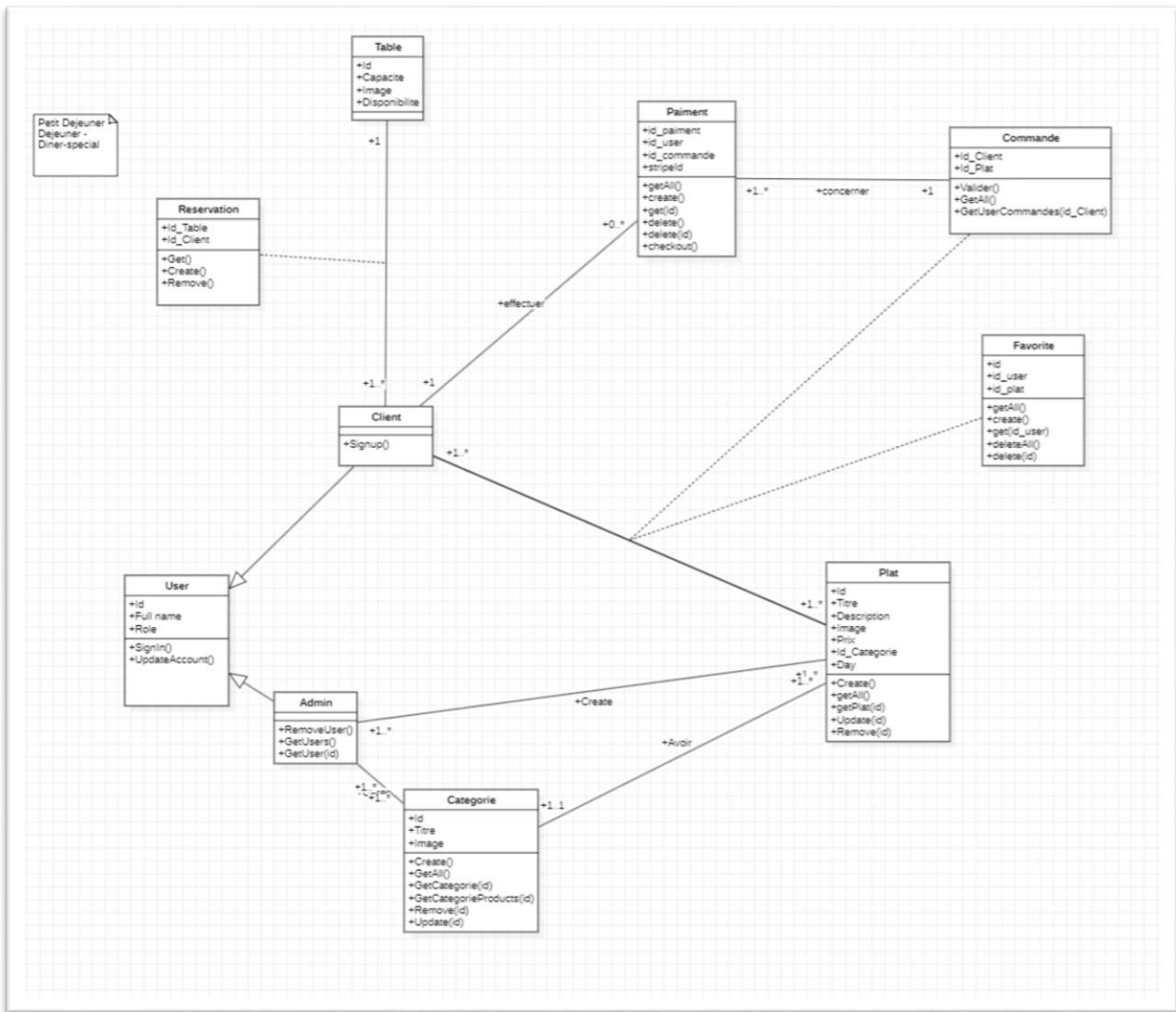
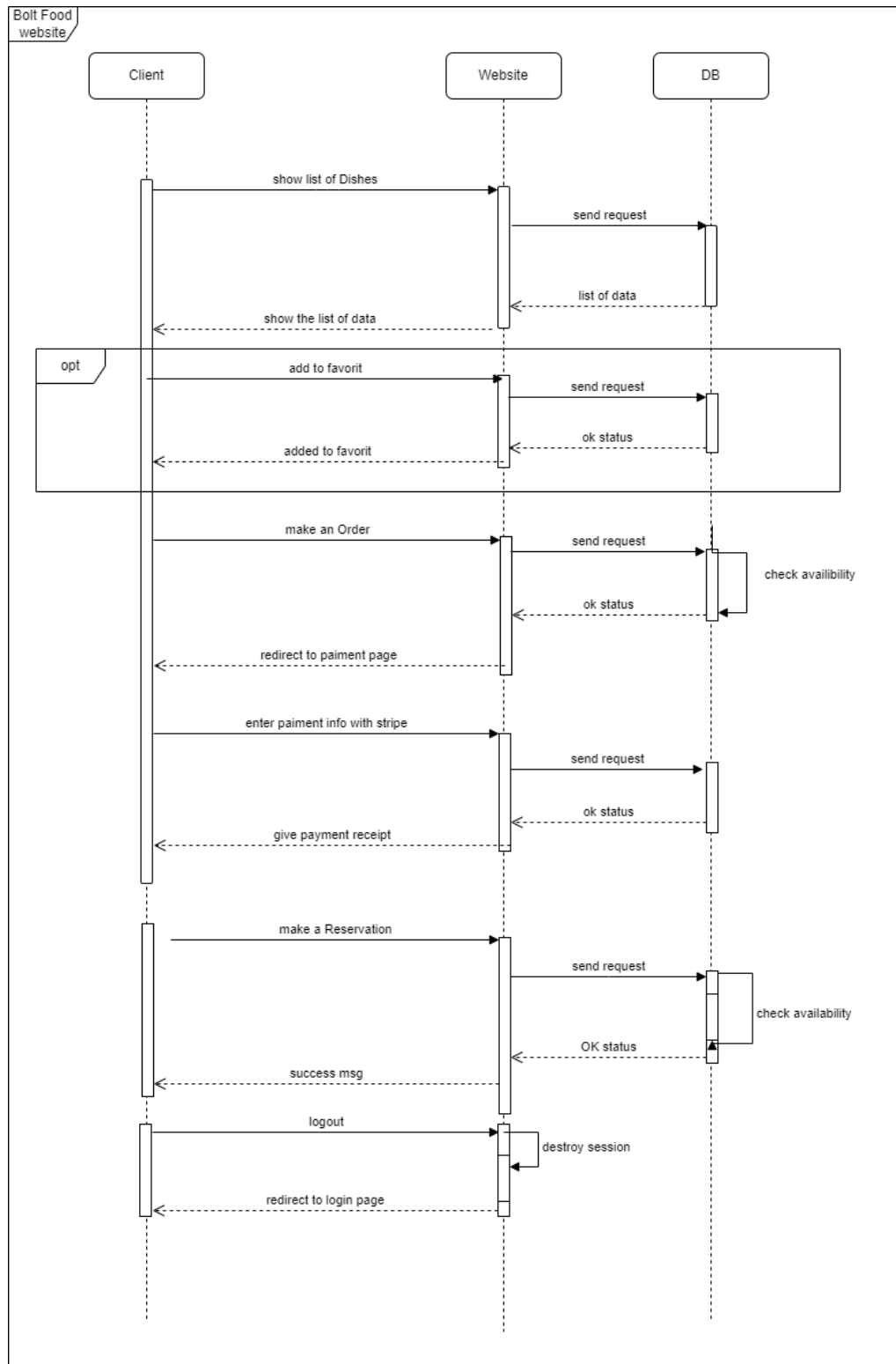
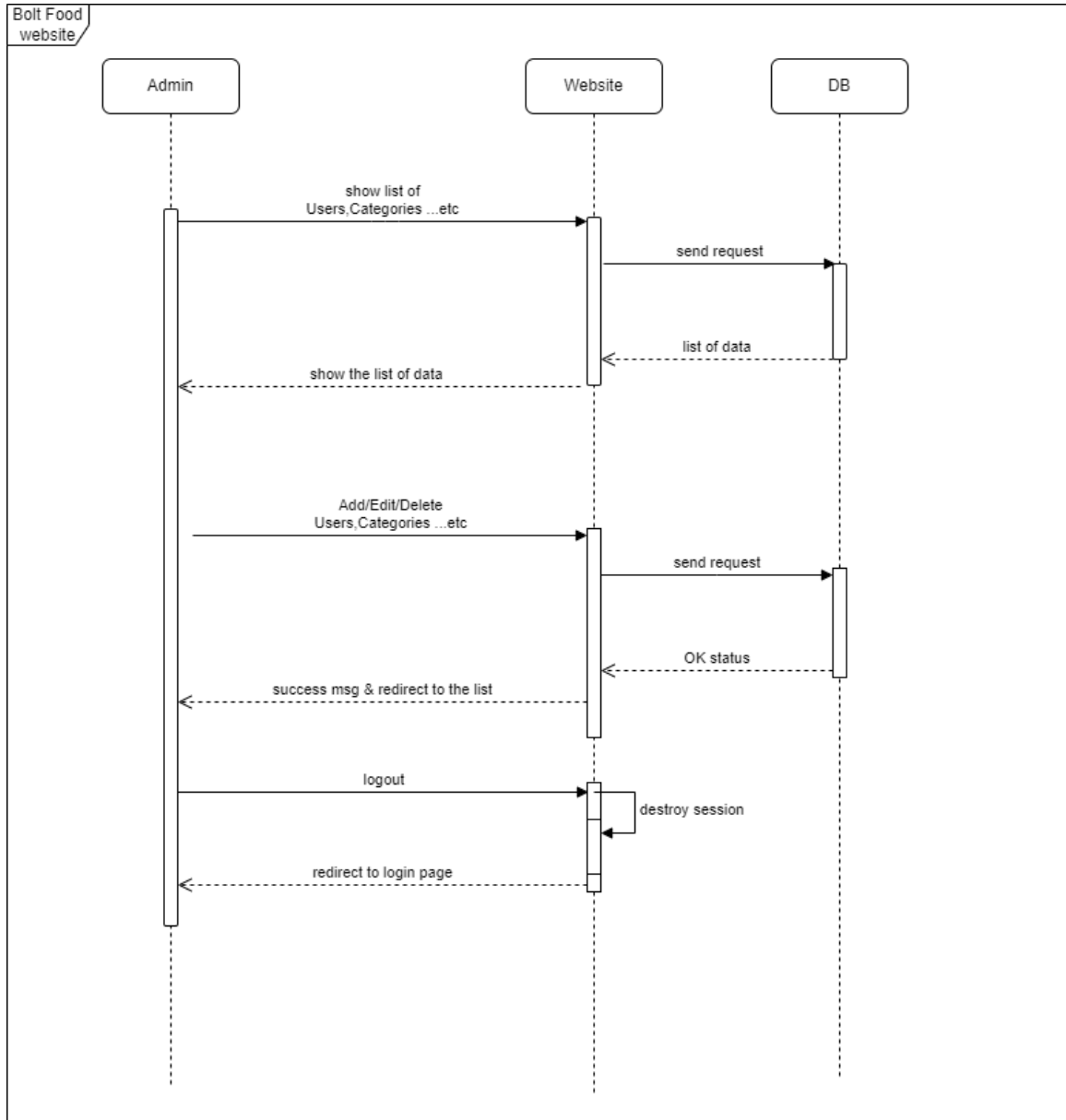


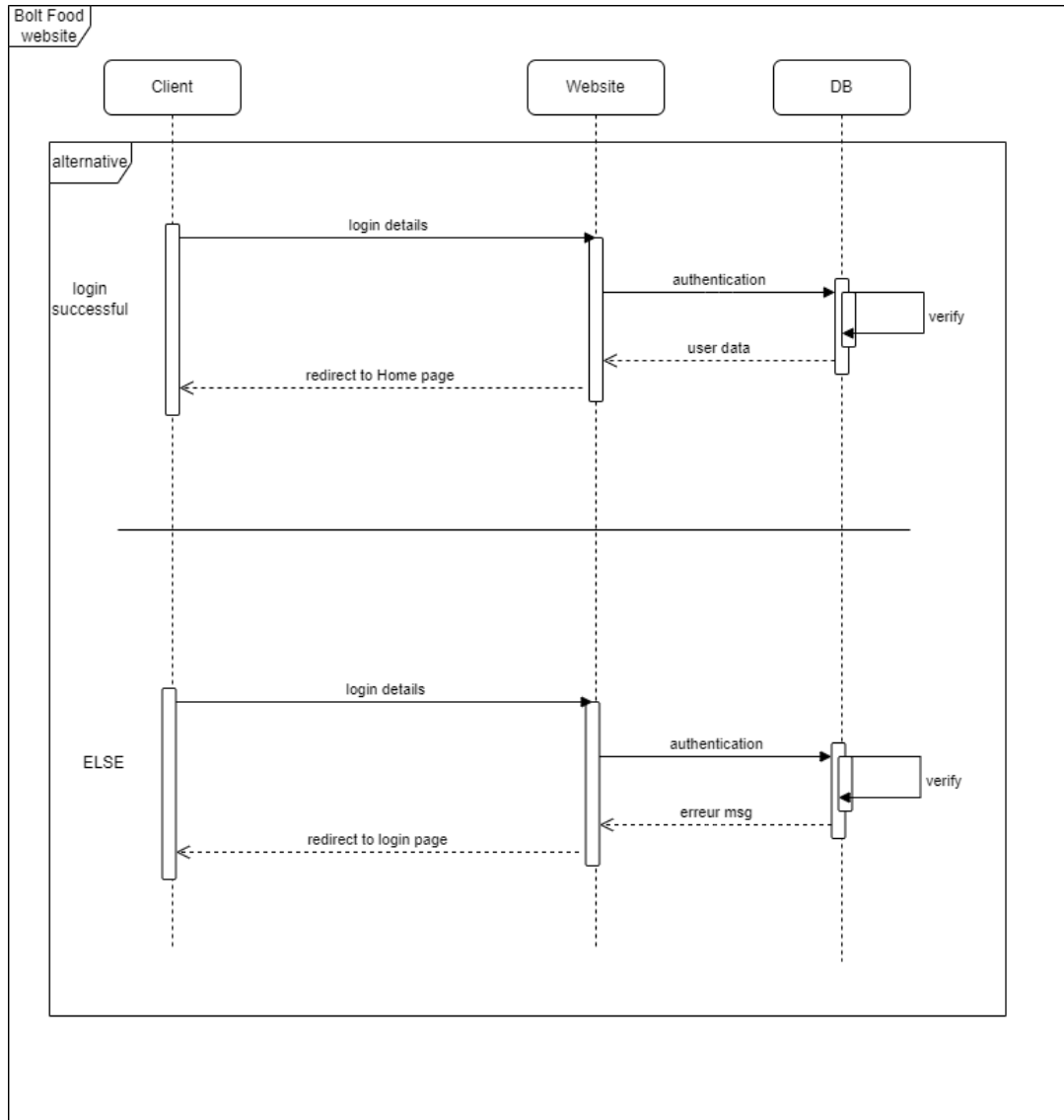
Figure 6: Diagramme de classes

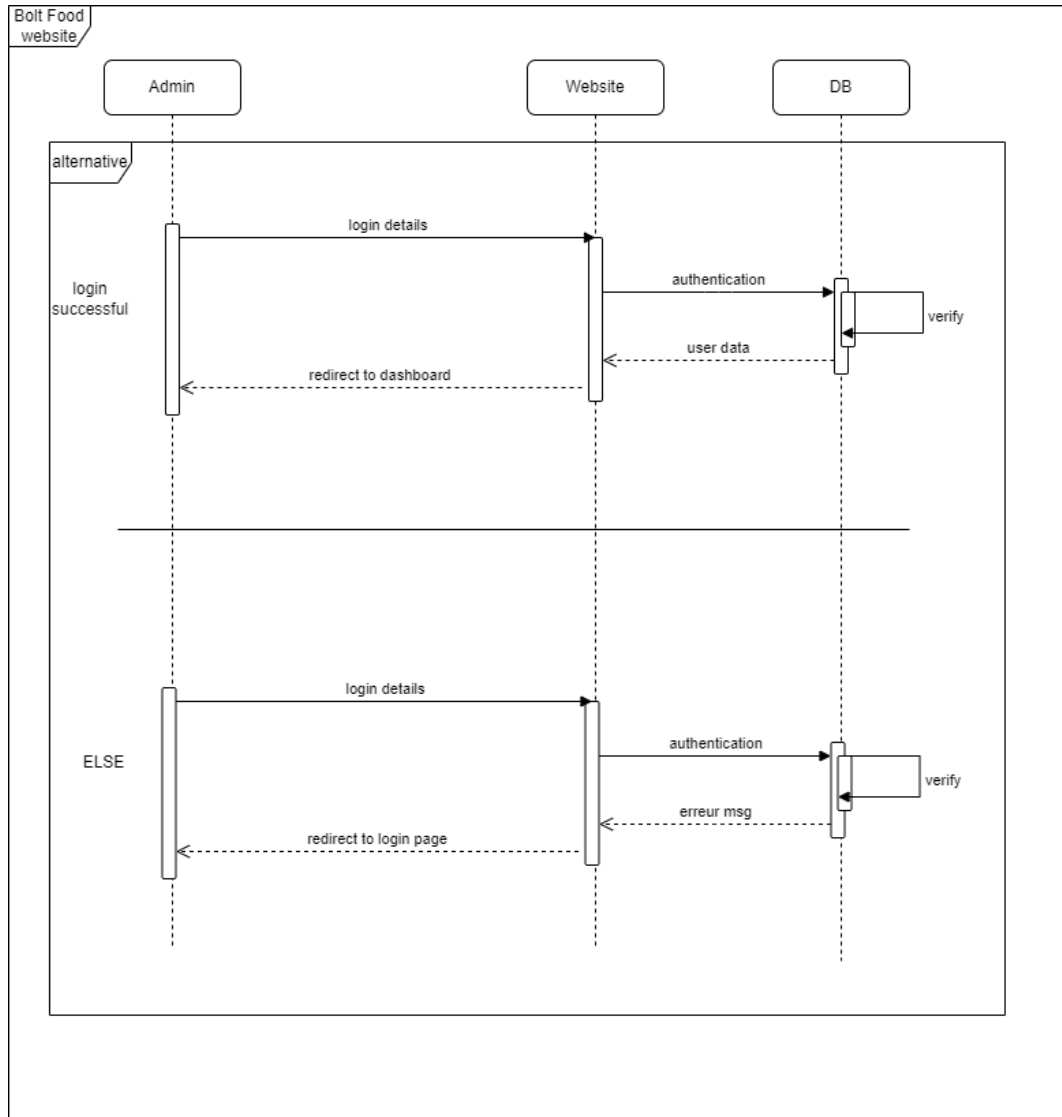
### III. Diagramme de sequences











## Conclusion

Dans ce chapitre, nous avons décrit la conception globale du système, à travers les différents diagrammes de UML ainsi qu'une description fonctionnelle du projet. Dans le chapitre suivant, nous abordons la phase de l'étude technique du projet.

---

# CHAPITRE 3: BASE DE DONNEES

---

## Introduction

Après avoir conçu le système d'information (diagramme de classes) et donc la base de données, il va maintenant falloir construire la base de données basée sur le système.

### C'est quoi base de données?

Une base de données est une collection organisée d'informations ou de données structurées, généralement stockées électroniquement dans un système informatique. Une base de données est généralement contrôlée par un système de gestion de base de données (SGBD).



### Choix du système de gestion de la base de données

Relationnelles	Non relationnelles
Les bases de données relationnelles sont devenues dominantes dans les années 1980. Les éléments d'une base de données relationnelle sont organisés sous forme d'un ensemble de tables avec des colonnes et des lignes. La technologie des bases de données relationnelles constitue le moyen le plus efficace et le plus flexible d'accéder à des informations structurées.	Une base de données NoSQL, ou base de données non relationnelle, permet de stocker et de manipuler des données non structurées et semi-structurées (contrairement à une base de données relationnelle). Les bases de données NoSQL sont devenues populaires à mesure que les applications Web devenaient plus courantes et plus complexes.
<ul style="list-style-type: none"> <li>• MySQL</li> <li>• Microsoft SQL Server</li> </ul>	<ul style="list-style-type: none"> <li>• Mongo DB</li> <li>• Cassandra</li> </ul>

Le système de gestion de bases de données (SGBD) a été axé sur SQL Server car il constitue une des conditions de travail sur le projet proposé par les professeurs, qui est le système de gestion de bases de données relationnelles développé par Microsoft.

## Implémentation de base de données

- Table Users

```
CREATE TABLE users (  
    id INT PRIMARY KEY IDENTITY(1,1),  
    email NVARCHAR(255) UNIQUE NOT NULL,  
    fname NVARCHAR(100) NOT NULL,  
    lname NVARCHAR(100) NOT NULL,  
    pass NVARCHAR(255) NOT NULL,  
    role NVARCHAR(50) NOT NULL  
);
```

- Table Categories

```
CREATE TABLE categories (  
    id INT PRIMARY KEY IDENTITY(1,1),  
    titre NVARCHAR(255) NOT NULL,  
    image NVARCHAR(255)  
);
```

- Table Dishes

```
CREATE TABLE dishes (  
    id INT PRIMARY KEY IDENTITY(1,1),  
    titre NVARCHAR(255) NOT NULL,  
    description NVARCHAR(MAX),  
    image NVARCHAR(255),  
    prix DECIMAL(10, 2) NOT NULL,  
    categorieId INT,  
    [day] NVARCHAR(50),  
    FOREIGN KEY (categorieId) REFERENCES categories(id)  
);
```

- Table Favorites

```
CREATE TABLE favorites (  
    favorite_id INT PRIMARY KEY IDENTITY(1,1),  
    clientId INT NOT NULL,  
    dishId INT NOT NULL,  
    FOREIGN KEY (clientId) REFERENCES users(id),  
    FOREIGN KEY (dishId) REFERENCES dishes(id)  
);
```

- Table Tables

```
CREATE TABLE tables (  
    id INT PRIMARY KEY IDENTITY(1,1),  
    capacite INT NOT NULL,  
    image NVARCHAR(255),  
    disponibilite BIT NOT NULL  
);
```

- Table Reservations

```
CREATE TABLE reservations (  
    id INT PRIMARY KEY IDENTITY(1,1),  
    clientId INT NOT NULL,  
    tableId INT NOT NULL,  
    reservation_date DATETIME DEFAULT GETDATE(),  
    FOREIGN KEY (clientId) REFERENCES users(id),  
    FOREIGN KEY (tableId) REFERENCES tables(id)  
);
```

- Table Orders

```
CREATE TABLE orders (  
  id INT PRIMARY KEY IDENTITY(1,1),  
  clientId INT NOT NULL,  
  dishId INT NOT NULL,  
  order_date DATETIME DEFAULT GETDATE(),  
  FOREIGN KEY (clientId) REFERENCES users(id),  
  FOREIGN KEY (dishId) REFERENCES dishes(id)  
);
```

- Table Paiements

```
CREATE TABLE Paiements (  
  paiement_id INT PRIMARY KEY IDENTITY(1,1),  
  clientId INT NOT NULL,  
  orderId INT NOT NULL,  
  stripId INT NOT NULL,  
  FOREIGN KEY (clientId) REFERENCES users(id),  
  FOREIGN KEY (orderId) REFERENCES orders(id)  
);
```

---

# CHAPITRE 4: OUTILS DE TRAVAUX

---



## Introduction

Ce chapitre explore les outils de travail clés utilisés pour atteindre les objectifs du projet. Nous examinerons de près les logiciels, équipements et méthodes employés, mettant en lumière leur contribution à notre réussite. Cette analyse nous permettra d'évaluer leur efficacité, d'identifier les points forts et les domaines d'amélioration, afin d'en tirer des enseignements pour les projets à venir.

## React

Nous avons opté pour React pour plusieurs raisons. Tout d'abord, React est largement reconnu comme l'un des frameworks front-end les plus populaires et les plus performants dans l'industrie du développement web. Sa popularité est due en partie à sa flexibilité, sa facilité d'utilisation et sa grande communauté de développeurs. De plus, React adopte une approche basée sur les composants, ce qui permet de créer des interfaces utilisateur modulaires et réutilisables, facilitant ainsi la maintenance et l'évolutivité des applications. En outre, la capacité de React à gérer efficacement les mises à jour de l'interface utilisateur grâce à sa virtual DOM offre des performances optimales, ce qui est essentiel pour offrir une expérience utilisateur fluide et réactive. (egacy.reactjs.org, 2020)



## React Router DOM

React Router DOM est une bibliothèque essentielle pour la navigation dans les applications web à page unique avec React. En définissant des routes correspondant à des URL spécifiques, elle permet de charger dynamiquement les composants associés. Vous pouvez également gérer la navigation programmée et les paramètres d'URL, offrant ainsi une expérience utilisateur fluide et dynamique. (reactrouter, 2020)



## Axios

Nous avons choisi Axios comme client HTTP pour interagir avec notre API RESTful depuis le côté client.

Axios est une bibliothèque JavaScript légère et simple à utiliser qui offre une syntaxe conviviale pour effectuer des requêtes HTTP asynchrones. Son support intégré pour les promesses permet une gestion efficace des appels réseau et des réponses asynchrones, ce qui est essentiel pour garantir la réactivité et la fluidité de l'interface utilisateur. De plus, Axios offre un large éventail de fonctionnalités avancées telles que la gestion des intercepteurs, la gestion des erreurs, et la configuration personnalisée des requêtes, ce qui en fait un choix idéal pour les applications web modernes. (axios, 2017)



## Redux

Redux est une bibliothèque JavaScript utilisée avec React pour gérer l'état global de l'application de manière prévisible. Avec Redux, vous définissez des actions pour décrire les changements d'état, des reducers pour spécifier comment ces changements sont appliqués, et un store centralisé pour stocker l'état global. Dans votre projet de plateforme de commerce avec chatbot intelligent, Redux vous aidera à maintenir un état cohérent et à gérer les interactions utilisateur de manière efficace. (redux, 2015)



## Postman

Postman est un outil essentiel pour les développeurs d'API, offrant des fonctionnalités de test, de débogage et de documentation. Il permet de tester facilement les API, d'automatiser les tests, de collaborer avec les membres de l'équipe et de garantir la sécurité des API. Dans le cadre de votre projet de plateforme de commerce avec chatbot intelligent, l'utilisation de Postman facilitera le développement, en assurant la qualité et la fiabilité des API sous-jacentes. (learning.postman.com, 2020)



## Stripe

Django Stripe est une intégration qui permet d'ajouter les fonctionnalités de paiement de Stripe à vos applications Django. Cette combinaison offre une gestion facile des paiements en ligne, y compris les paiements uniques, les abonnements et les remboursements. Avec Django Stripe, vous pouvez sécuriser et gérer efficacement les transactions financières dans vos applications Django. (docs.stripe, 2016)



## Express

Express.js, parfois aussi appelé « Express », est un framework backend Node.js minimaliste, rapide et de type Sinatra qui offre des fonctionnalités et des outils robustes pour développer des applications backend évolutives. Il vous offre le système de routage et des fonctionnalités simplifiées pour étendre le framework en développant des composants et des parties plus puissants en fonction des cas d'utilisation de votre application. (kinsta, 2022)



## Microsoft SQL Server

Microsoft SQL Server est un système de gestion de base de données relationnelle développé par Microsoft. Depuis sa création en 1989, SQL Server a évolué pour devenir une véritable plateforme d'informations d'entreprise pour servir un large éventail d'applications. Il est doté d'un ensemble d'outils permettant la gestion et l'administration d'une base de données, la programmation à travers T-SQL (Transact-SQL), sa propre implémentation du langage SQL, la Business Intelligence et l'analyse des données ainsi que le développement d'applications. (datascientest, 2023)



## GitHub

GitHub est une plateforme de développement collaboratif basée sur Git, permettant le stockage centralisé du code source, la collaboration entre les membres de l'équipe via les pull requests et les issues, le suivi des modifications, l'automatisation du déploiement avec GitHub Actions, et la gestion de projet avec des fonctionnalités telles que les tableaux Kanban. Son utilisation dans votre projet de plateforme de commerce avec chatbot intelligent facilite la gestion du code, la collaboration et le déploiement continu de votre application. (docs.github, 2016)



## Visual Studio Code

Visual Studio Code (VS Code) est un éditeur de code gratuit et open-source développé par Microsoft. Il offre une expérience de codage fluide avec une coloration syntaxique, une saisie semi-automatique et une intégration Git native. VS Code est hautement extensible grâce à un large éventail d'extensions et offre des outils de débogage et un terminal intégré. Son utilisation dans votre projet de plateforme de commerce avec chatbot intelligent peut améliorer la productivité et simplifier le développement. (code.visualstudio, 2018)



## Conclusion

En conclusion, ce chapitre a souligné l'importance des outils de travail pour notre projet. En identifiant les points forts et les domaines à améliorer, nous sommes mieux préparés pour nos futures initiatives.

---

# CHAPITRE 5 : MISE EN OEUVRE ET REALISATION

---

## Introduction

Cette partie présente les étapes de développement de ce projet. Le modèle relationnel implémenté du côté de la base de données, et la présentation du framework **React js** et **Express.js** côté développement web. Nous citerons également quelques exemples d'interfaces web de l'application.

### I. Server (Back End)

#### a. Presentation du framework express js

Aujourd'hui, l'utilisation d'un Framework est devenue obligatoire dans le monde professionnel du développement web. Les Framework offrent un mode de travail structuré permettant de maintenir facilement un projet.

##### *Pourquoi Express js?*

Ce framework fournit un ensemble d'outils pour les applications web, les requêtes et les réponses HTTP, le routage et les intergiciels permettant de créer et de déployer des applications à grande échelle, prêtes pour l'entreprise. (kinsta, 2022)



##### *Rôle de Express*

Dans ce projet, nous ne voulons rien afficher en vue express, toutes les données seront envoyées par api, pour une utilisation dans une autre application Web développée par react js, elle gèrera la partie conception.

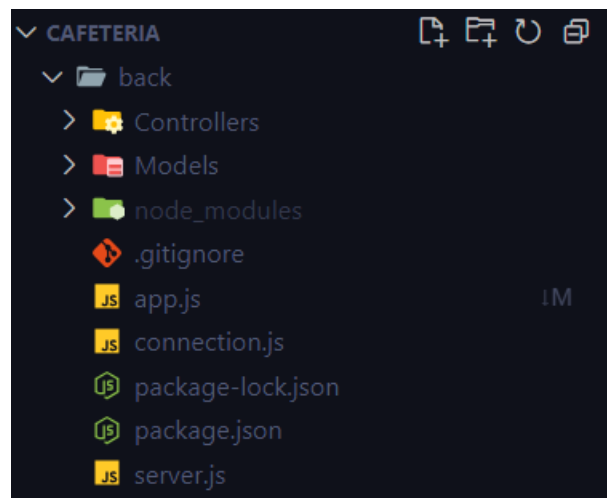
##### *L'application Express.js*

Maintenant, créons une application de démonstration du monde réel en utilisant le nouveau Express.js 4.19. Pour commencer, créez un répertoire pour l'application et installez les paquets suivants :

```
mkdir nom-application  
cd nom-application  
npm install express nodemon cors body-parser--save
```

## Structure de l'application

Un projet Express est constitué de plusieurs dossiers et fichiers comme dans la figure:



## Configuration

Le fichier connection.js contient toute la configuration du connection avec la base de donnes SQL Server. Il définit les paramètres de connexion à la base de données.

```

back > js connection.js > ...
  1  const config = {
  2      user: 'cafeteria',
  3      password: 'cafeteria',
  4      server: 'localhost\\SQLEXPRESS',
  5      database: 'cafeteria',
  6      options: {
  7          encrypt: true,
  8          enableArithAbort: true,
  9          trustServerCertificate: true
 10      }
 11  };
 12
 13  module.exports = config;
  
```

### a. Mise en place du backend de conception-MVC (**voir annexe**)

Element	
<b>Le contrôleur</b>	<p>Un controller est une classe qui va contenir différentes méthodes. Chaque méthode correspondant généralement à une opération (URL) de l'application.</p> <p>Dans le modèle MVC (modèle-vue-contrôleur), le contrôleur contient la logique concernant les actions effectuées par l'utilisateur. En pratique, dans une application express, l'utilisation de contrôleurs permet de libérer les routes du code qu'elles contiennent dans leurs fonctions de rappel.</p>
<b>Le modèle</b>	<p>Un modèle est une classe métier représentant une partie des données d'une application.</p>



	Dans la plupart des cas, un modèle est associé à une table de la base de données. Model est la classe de base des modèles d'une application. Cette classe met à disposition des fonctionnalités <b>CRUD</b> (pour Create, Read, Update, Delete), offre des possibilités de recherche avancées et permet de gérer les relations entre modèles, le tout sans avoir besoin d'utiliser SQL.
<b>La vue</b>	Les vues dans express contiennent une logique qui détermine comment présenter le résultat final de l'application Web aux utilisateurs. La sortie est souvent la sortie HTML qui, après chaque réponse, revient aux utilisateurs.
<b>L'api</b>	Pour les applications web, une API (Application Programming Interface) ou « Interface de Programmation d'Applications » en français, est un ensemble de définitions et de protocoles qui permettent à des applications de communiquer et de s'échanger mutuellement des données ou des services.

## b. Rôle de Express

Dans ce projet, nous ne voulons rien afficher en vue express, toutes les données seront envoyées par api, pour une utilisation dans une autre application Web développée par react js, elle gèrera la partie conception.

## c. Fonctionnement du serveur

On démarre le serveur en exécutant la commande **npm start**, qui exécute la commande **nodemon server** comme raccourci.

```
npm start / nodemon server
```

Ensuite nous pouvons utiliser les requêtes **HTTP** du serveur (**CRUD**).

## d. Les classes

### *Dishe*

```
async getAllDishes(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("select * from dishes", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async createDish(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("INSERT INTO dishes (titre,description,image, prix, categorieId,day) VALUES ('${req.body.titre}', '${req.body.description}', '${req.body.image}', ${req.body.prix}, ${req.body.categorieId}, ${req.body.day})", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async getDishById(req, res, next) {
  try {
    const dishId = req.params.dishId;

    let request = new sql.Request();
    request.query(`select * from dishes where id=${dishId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async updateDish(req, res, next) {
  try {
    const dishId = req.params.dishId;

    const updateOps = {};
    for (const ops of req.body) {
      updateOps[ops.propName] = ops.value;
    }

    res.status(200).json({ status: "success", message: "User updated" });
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteDish(req, res, next) {
  try {
    const dishId = req.params.dishId;

    let request = new sql.Request();
    request.query(`delete from dishes where id='${dishId}'`, (err, records) => {
      if (err) {
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteAllDishes(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`delete from dishes`, (err, records) => {
      if (err) {
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success" });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

## Categorie

```

async createCategory(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`INSERT INTO categories (titre,image) VALUES ('${req.body.titre}', '${req.body.image}')`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async deleteAllCategories(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`delete from categories`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success"});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async deleteCategory(req, res, next) {
  try {
    const categoryId = req.params.categoryId;

    let request = new sql.Request();
    request.query(`delete from categories where id='${categoryId}'`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0]});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async getAllCategories(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("select * from categories", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async getCategoryById(req, res, next) {
  try {
    const categoryId = req.params.categoryId;

    let request = new sql.Request();
    request.query(`select * from categories where id=${categoryId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async updateCategory(req, res, next) {
  try {
    const categoryId = req.params.categoryId;

    const updateOps = {};
    for (const ops of req.body) {
      updateOps[ops.propName] = ops.value;
    }

    res.status(200).json({ status: "success", message: "User updated" });
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

## Favorite

```
✓ class FavoritController {  
✓   constructor() {  
    this.initializeRoutes();  
  }  
  
✓   initializeRoutes() {  
    this.router = router;  
    router.get('/', this.getAllFavorits.bind(this));  
    router.post('/', this.createFavorit.bind(this));  
    router.get('/:userId', this.getFavoritByClient.bind(this));  
    router.delete('/:favoritId', this.deleteFavorit.bind(this));  
    router.delete('/', this.deleteAllFavorits.bind(this));  
  }  
}
```

```
async getAllFavorits(req, res, next) {  
  try {  
    let request = new sql.Request();  
    request.query("select * from favorites", (err, records)=> {  
      if(err) console.log(err);  
      else {  
        res.status(200).json({ status: "success", data: records.recordsets[0] });  
      }  
    })  
  } catch (error) {  
    console.log(error);  
    res.status(500).json({ error: error });  
  }  
}
```

```
async createFavorit(req, res, next) {  
  try {  
    let request = new sql.Request();  
    request.query(`INSERT INTO favorites (clientId,dishId) VALUES ('${req.body.clientId}', '${req.body.dishId})`  
    if(err) console.log(err);  
    else {  
      res.status(200).json({ status: "success", data: records });  
    }  
  })  
} catch (error) {  
  console.log(error);  
  res.status(500).json({ error: error });  
}
```

```
async getFavoritByClient(req, res, next) {
  try {
    const userId = req.params.userId;

    let request = new sql.Request();
    request.query(`select * from favorsits where userid=${userId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteFavorit(req, res, next) {
  try {
    const favoritId = req.params.favoritId;

    let request = new sql.Request();
    request.query(`delete from favorsits where id='${favoritId}'`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0]});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteAllFavorits(req, res, next) {  
  try {  
    let request = new sql.Request();  
    request.query(`delete from favorits`, (err, records)=> {  
      if(err){  
        res.status(400).json();  
        console.log(err);  
      } else {  
        res.status(200).json({ status: "success"});  
      }  
    })  
  } catch (error) {  
    console.log(error);  
    res.status(500).json({ error: error });  
  }  
}
```



## Table

```
class TableController {
  constructor() {
    this.initializeRoutes();
  }

  initializeRoutes() {
    this.router = router;
    router.get('/', this.getAllTables.bind(this));
    router.post('/', this.createTable.bind(this));
    router.get('/:tableId', this.getTableById.bind(this));
    router.patch('/:tableId', this.updateTable.bind(this));
    router.delete('/:tableId', this.deleteTable.bind(this));
    router.delete('/', this.deleteAllTables.bind(this));
  }
}
```

```
async getAllTables(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("select * from tables", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async createTable(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`INSERT INTO tables (capacite,image, disponibilite) VALUES ('${req.body.capacite}', '${req.b
    if(err) console.log(err);
    else {
      res.status(200).json({ status: "success", data: records });
    }
  })
} catch (error) {
  console.log(error);
  res.status(500).json({ error: error });
}
```

```
async getTableById(req, res, next) {
  try {
    const tableId = req.params.tableId;

    let request = new sql.Request();
    request.query(`select * from tables where id=${tableId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async updateTable(req, res, next) {
  try {
    const tableId = req.params.tableId;

    const updateOps = {};
    for (const ops of req.body) {
      updateOps[ops.propName] = ops.value;
    }
    res.status(200).json({ status: "success", message: "User updated" });
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteTable(req, res, next) {
  try {
    const tableId = req.params.tableId;

    let request = new sql.Request();
    request.query(`delete from tables where id='${tableId}'`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0]});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteAllTables(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`delete from tables`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success"});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

## Users

```
async deleteAllUsers(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`delete from users`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success"});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteUser(req, res, next) {
  try {
    const userId = req.params.userId;

    let request = new sql.Request();
    request.query(`delete from users where id='${userId}'`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0]});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async updateUser(req, res, next) {
  try {
    const userId = req.params.userId;

    const updateOps = {};
    for (const ops of req.body) {
      updateOps[ops.propName] = ops.value;
    }

    res.status(200).json({ status: "success", message: "User updated" });
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async loginUser(req, res, next) {
  try {
    const email = req.body.email;
    const pass = req.body.pass;

    let request = new sql.Request();
    request.query(`select * from users where email='${email}' and pass='${pass}'`, (err, records) => {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordset[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async getUserById(req, res, next) {
  try {
    const userId = req.params.userId;

    let request = new sql.Request();
    request.query(`select * from users where id=${userId}`, (err, records) => {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```

async createUser(req, res, next) {
  try {
    let request = new sql.Request();
    request.query('INSERT INTO users (email, fname, lname, pass, role) VALUES (${req.body.email}, ${req.body.fname}',
      if(err) console.log(err);
    else {
      res.status(200).json({ status: "success", data: records });
    }
  })
} catch (error) {
  console.log(error);
  res.status(500).json({ error: error });
}
}

```

```

async getAllUsers(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("select * from users", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

## Reservation

```

class ReservationController {
  constructor() {
    this.initializeRoutes();
  }

  initializeRoutes() {
    this.router = router;
    router.get('/', this.getAllReservations.bind(this));
    router.post('/', this.createReservation.bind(this));
    router.get('/:reservationId', this.getReservationById.bind(this));
    router.delete('/:reservationId', this.deleteReservation.bind(this));
  }
}

```

```
async getAllReservations(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("select * from reservations", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async createReservation(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`INSERT INTO reservations (clientId,tableId) VALUES ('${req.body.clientId}', '${req.body.tableId}')`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async getReservationById(req, res, next) {
  try {
    const reservationId = req.params.reservationId;

    let request = new sql.Request();
    request.query(`select * from reservations where id=${reservationId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```

async deleteReservation(req, res, next) {
  try {
    const reservationId = req.params.reservationId;

    let request = new sql.Request();
    request.query(`delete from reservations where id='${reservationId}'`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0]});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

## Order

```

async getAllOrders(req, res, next) {
  try {
    let request = new sql.Request();
    request.query("select * from orders", (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async createOrder(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`INSERT INTO orders (clientId,dishId) VALUES ('${req.body.clientId}', '${req.body.dishId}')`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```



```

async getOrderById(req, res, next) {
  try {
    const orderId = req.params.orderId;

    let request = new sql.Request();
    request.query(`select * from orders where id=${orderId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async getOrderByClient(req, res, next) {
  try {
    const userId = req.params.userId;

    let request = new sql.Request();
    request.query(`select * from orders where clientId=${userId}`, (err, records)=> {
      if(err) console.log(err);
      else {
        res.status(200).json({ status: "success", data: records.recordsets[0] });
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```

async updateOrder(req, res, next) {
  try {
    const orderId = req.params.orderId;

    const updateOps = {};
    for (const ops of req.body) {
      updateOps[ops.propName] = ops.value;
    }

    res.status(200).json({ status: "success", message: "Order updated" });
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}

```

```
async deleteOrder(req, res, next) {
  try {
    const orderId = req.params.orderId;

    let request = new sql.Request();
    request.query(`delete from orders where id='${orderId}'`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success", data: records.recordsets[0]});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

```
async deleteAllOrders(req, res, next) {
  try {
    let request = new sql.Request();
    request.query(`delete from orders`, (err, records)=> {
      if(err){
        res.status(400).json();
        console.log(err);
      } else {
        res.status(200).json({ status: "success"});
      }
    })
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: error });
  }
}
```

## Conclusion

Cette partie a été consacrée à la présentation de la méthode de développement qui a été menée dans ce projet, et les outils qui nous ont servi d'appui pour trouver des solutions à la problématique qui a été posée au début du projet, afin de satisfaire les besoins des utilisateurs.

## Conclusion et perspective

Le projet de gestion de cafétéria consiste en la conception et le développement d'un système informatisé complet destiné à rationaliser toutes les activités opérationnelles d'une cafétéria. Ce système sera doté de fonctionnalités détaillées, telles que la gestion détaillée des stocks incluant la gestion des fournisseurs, la réception des marchandises et le suivi des niveaux de stock en temps réel. Il permettra également la prise de commandes efficace, en offrant aux clients la possibilité de passer des commandes en ligne ou sur place, tout en garantissant une gestion précise des paiements et des transactions.

De plus, le système inclura un module de gestion des employés permettant de gérer les horaires de travail, les affectations de tâches et le suivi des performances individuelles. Les fonctionnalités de reporting seront également intégrées pour générer des rapports détaillés sur les ventes, les tendances de consommation, les niveaux de stock et les performances du personnel. Ces rapports seront essentiels pour prendre des décisions stratégiques et améliorer continuellement les opérations de la cafétéria.

En outre, le système sera conçu de manière conviviale, avec une interface utilisateur intuitive pour faciliter son utilisation par le personnel de la cafétéria. Des mesures de sécurité robustes seront mises en place pour garantir la protection des données sensibles et des transactions financières. Dans l'ensemble, ce système de gestion de cafétéria vise à améliorer l'efficacité opérationnelle, à réduire les coûts, à minimiser les pertes de stocks et à offrir une expérience client optimale.

## Annexe

### I. L'architecture MVC et l'utilisation des Frameworks

#### a. Les Frameworks

Un framework (ou infrastructure logicielle en français) désigne en programmation informatique un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application, encore appelé structure logicielle, canevas ou socle d'applications en français, qui établit les fondations d'un logiciel ou son squelette applicatif. Tous les développeurs qui l'utilisent peuvent l'enrichir pour en améliorer l'utilisation.

##### Avantages de l'utilisation des framework

- L'objectif premier d'un framework est d'améliorer la productivité des développeurs qui l'utilisent en offrant des briques prêtes à être utilisées, autrement dit, le framework s'occupe de la forme et permet au développeur de se concentrer sur le fond.
- Un framework améliore la façon de travailler, en utilisant l'architecture MVC, qui permet d'organiser le code des développeurs.
- Une communauté active, une documentation de qualité et régulièrement mise à jour.

##### Inconvénients

- Une courbe d'apprentissage élevée. En effet, pour maîtriser un framework, il faut un temps d'apprentissage non négligeable. Chaque brique qui compose un framework a sa complexité et d'autre part des connaissances préalables du Design pattern.

#### b. L'architecture MVC

L'architecture Modèle/Vue/Contrôleur (MVC) est une façon d'organiser une interface graphique d'un programme. Elle consiste à distinguer trois entités distinctes qui sont, le modèle, la vue et le contrôleur ayant chacun un rôle précis dans l'interface.

L'organisation globale d'une interface graphique est souvent délicate. Bien que la façon MVC d'organiser une interface ne soit pas la solution miracle, elle fournit souvent une première approche qui peut ensuite être adaptée. Elle offre aussi un cadre pour structurer une application.

Dans l'architecture MVC, les rôles des trois entités sont les suivants :

- **Modèle** : données (accès et mise à jour).
  - **Vue** : interface utilisateur (entrées et sorties).
  - **Contrôleur** : gestion des événements et synchronisation.
- 
- **Modèle** : Son rôle est d'aller récupérer les informations brutes dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur.
  - **Vue** : S'occupe des interactions avec l'utilisateur, la présentation, la saisie et la validation des données. On y trouve essentiellement du code HTML mais aussi quelques boucles et conditions.
  - **Contrôleur** : cette partie gère la dynamique de l'application. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue.

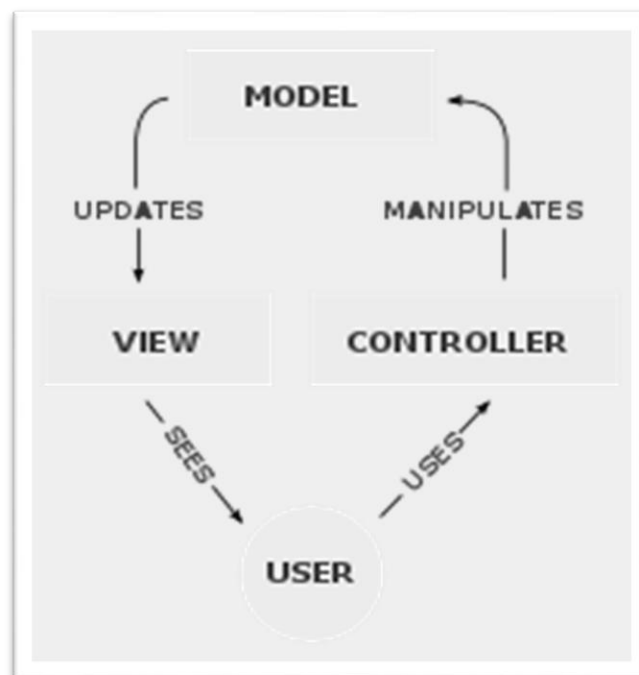


Figure 7: : Interactions entre le modèle, la vue et le contrôleur

## II. Gestion des produits avec Postman

### 1. La page principale de Postman

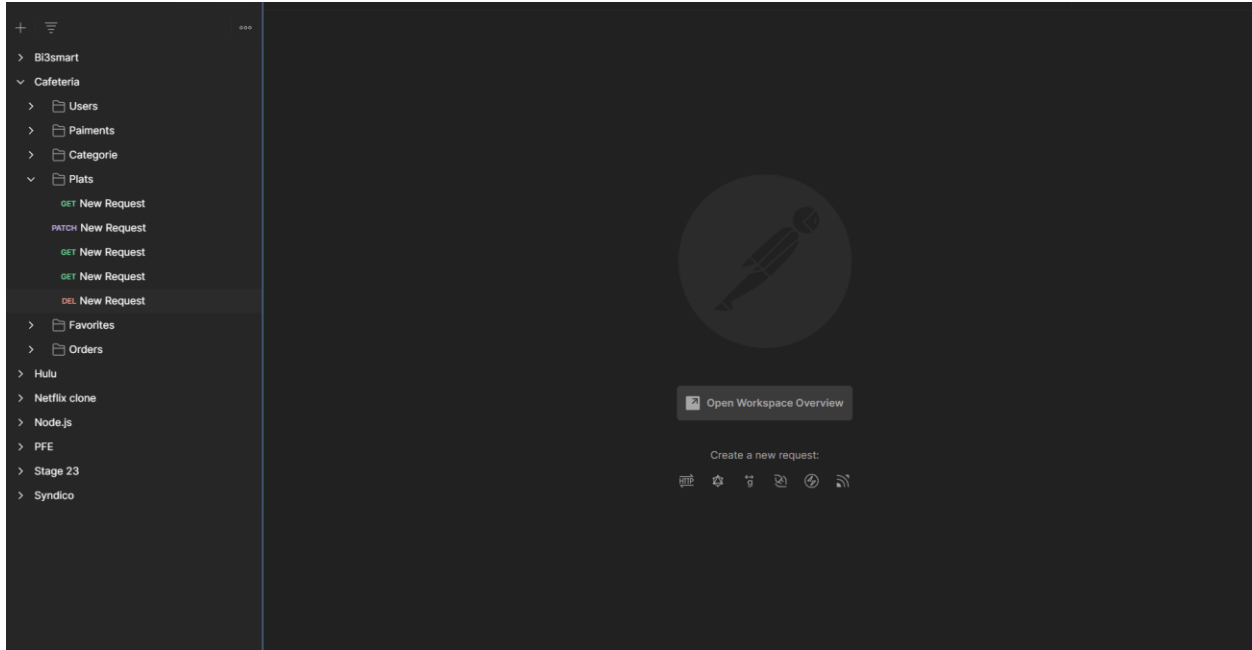


Figure 8: Page d'accueil du postman.

### 2. Comment fonctionne Postman?

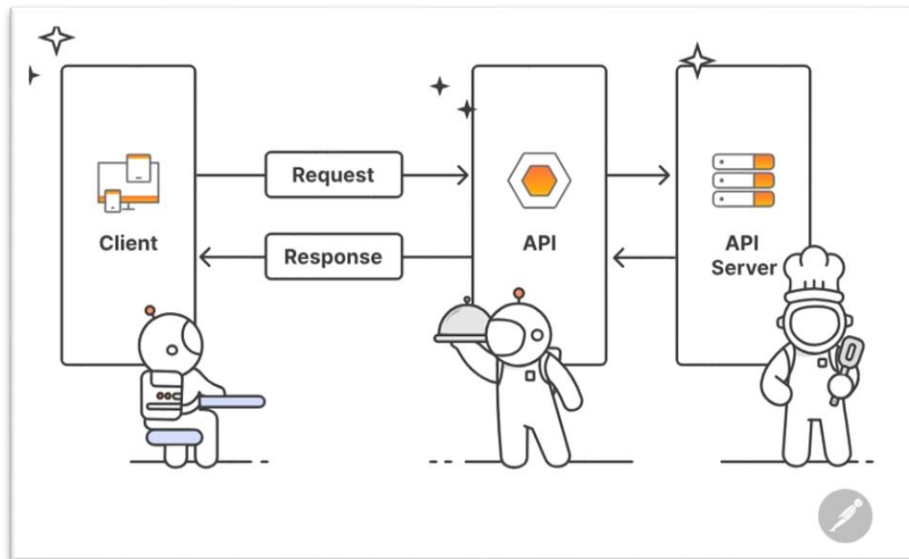


Figure 9: Comment fonctionne Postman?

## HTTP Requests

Titre	Request	Link
All plats	Get	http://127.0.0.1:8000/api/plats/
Obtenir les plats par catégorie	Get	http://localhost:3005/plats/categorie/1
Create plat	Post	http://127.0.0.1:8000/api/plats/
Update plat	Patch	http://127.0.0.1:8000/api/plats/7
Delete plat	Delete	http://127.0.0.1:8000/api/plats/7

Tableau 1: Requets Produits

## Responses

### 1. All plats

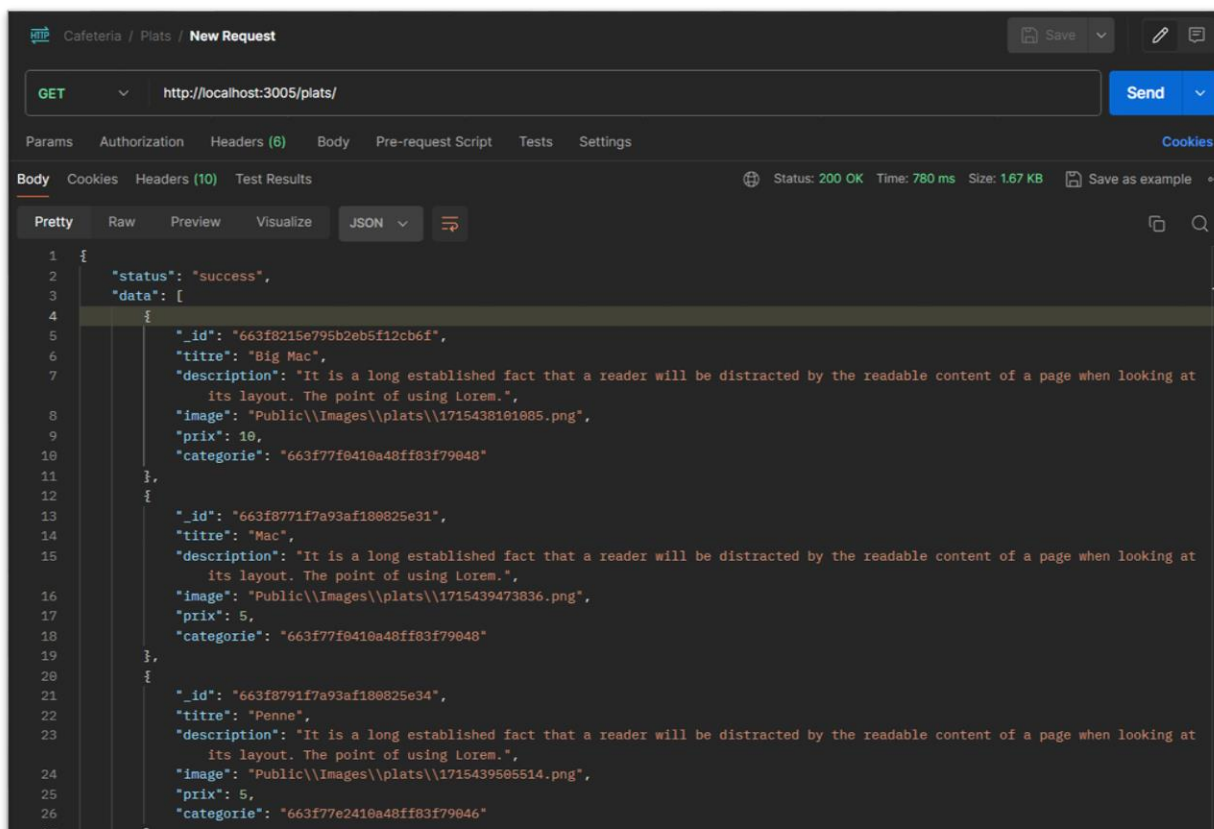


Figure 10: Requet http "GET"

## 2. Create Product

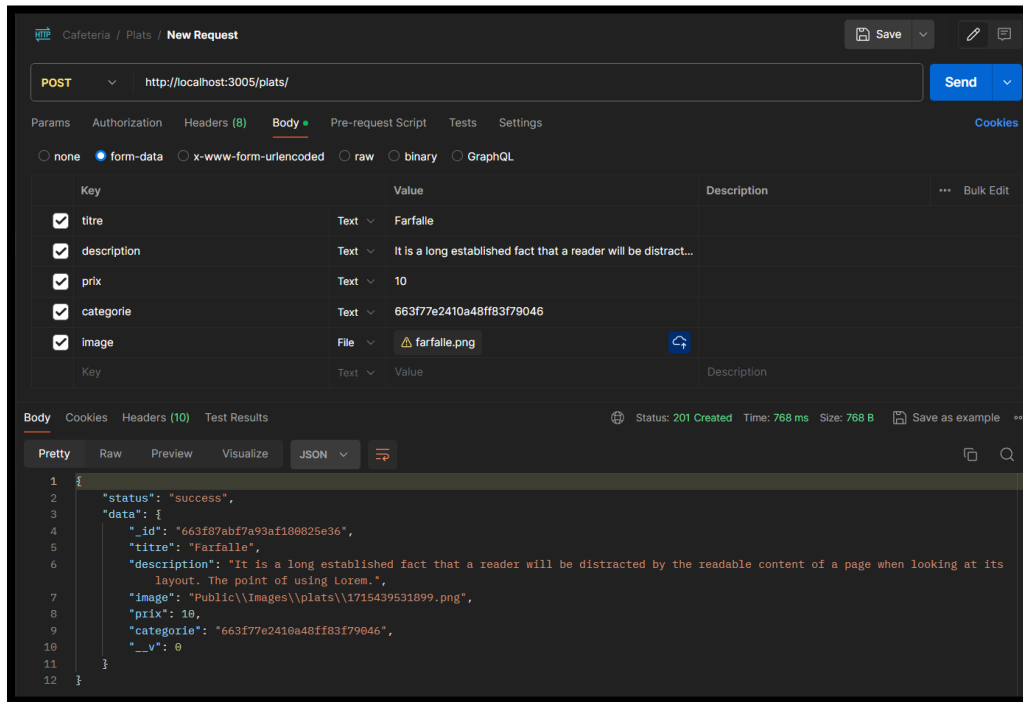


Figure 11: Requet http "POST"

## 3. Update Product

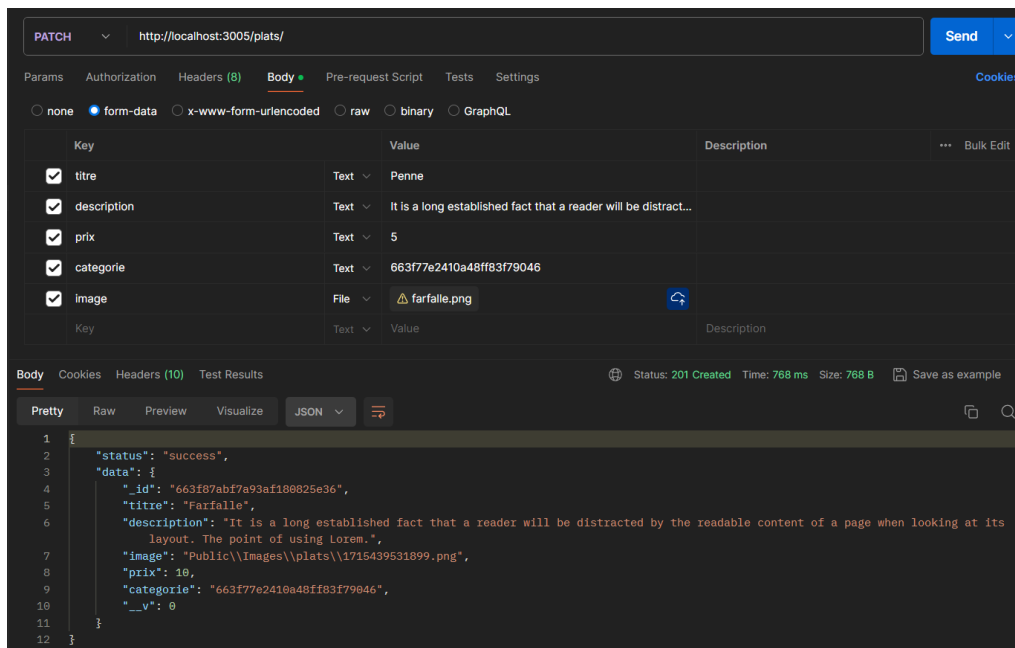


Figure 12: Requet http "PATCH"



## 4. Delete Product

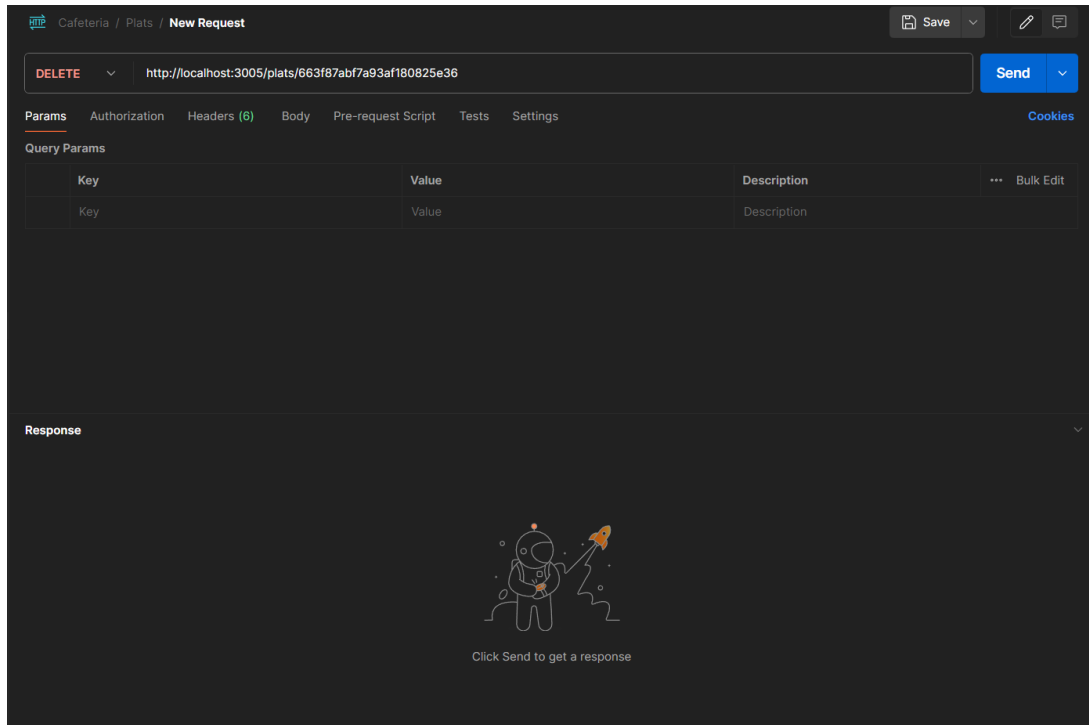


Figure 13: Requet http "DELETE"

### III. Comment la Commande travailler sous couverture

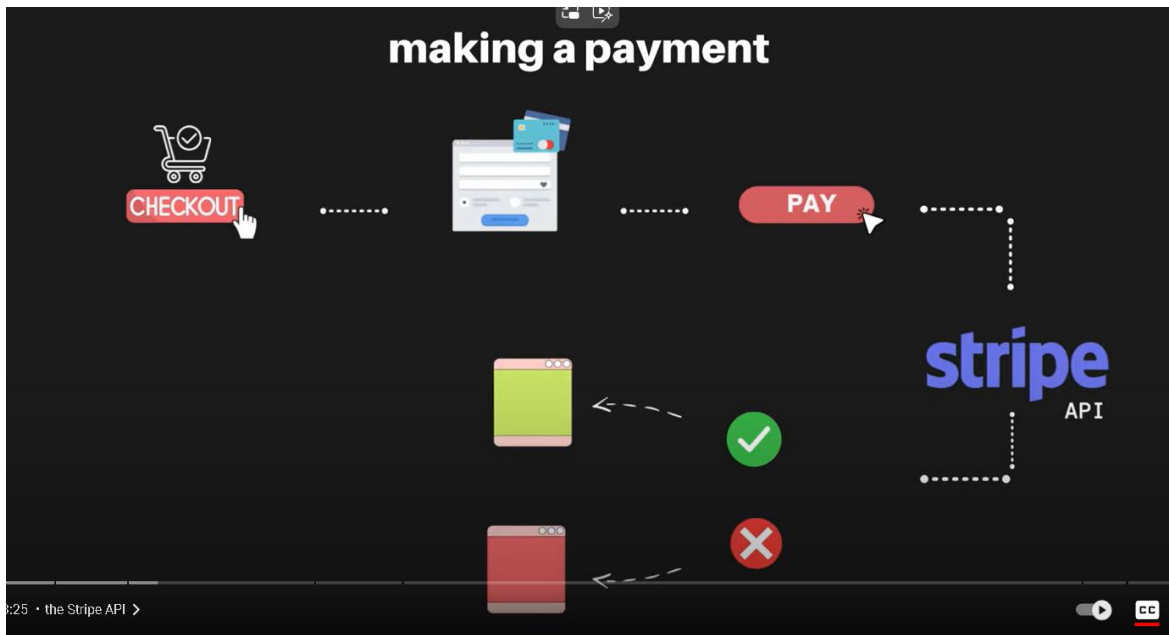


Figure 14: Comment fonctionne le commande ?

#### a. Valider le paiement

Après commander le client sera redirigé vers la page de paiement propulsée par Stripe, le client choisira donc de payer par carte ou paypal, puis il renseignera ses informations.

The screenshot shows the Stripe payment interface. On the left, a summary of the order is displayed: **Pay \$300.00**. Below this, a list of items is shown: 'Qty 1' for \$100.00 and 'Qty 2' for \$200.00, totaling \$300.00. On the right, the payment method is selected as 'Card'. The card information is entered: card number '4242 4242 4242 4242', expiry date '04 / 44', and cardholder name 'Soulimane ouhmda'. The country is set to 'Morocco'. A 'Processing...' button is at the bottom right.

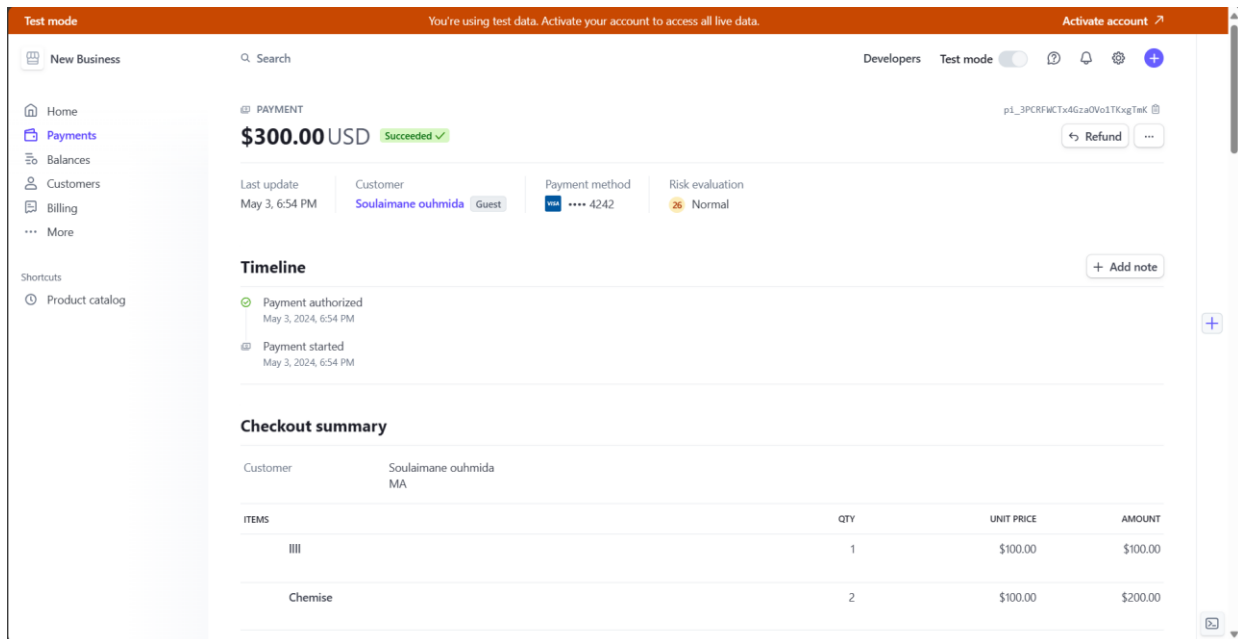
Figure 15: la page de paiement

## b. Consulter la commande

Après cela, le moteur Stripe traitera le paiement.

- ❌ S'il n'est pas correct, le client sera redirigé vers la page de la commande.
- ✅ Si tout se passe bien, la commande sera créée dans la base de données et le client sera redirigé vers la page des commandes, afin qu'il puisse voir sa commande.

Voici les détails de la commande sur le site Stripe :



The screenshot shows the Stripe dashboard interface. At the top, it indicates 'Test mode' and 'You're using test data. Activate your account to access all live data.' The main section displays a successful payment of \$300.00 USD. Below this, there is a timeline showing 'Payment authorized' and 'Payment started' on May 3, 2024, at 6:54 PM. A 'Checkout summary' section lists the customer as Soulaimane ouhmida (MA) and details the items purchased: 1 unit of 'III' at \$100.00 and 2 units of 'Chemise' at \$200.00.

ITEMS	QTY	UNIT PRICE	AMOUNT
III	1	\$100.00	\$100.00
Chemise	2	\$100.00	\$200.00

Figure 16: La commande sur Stripe.

## Bibliographie

- axios*. (2017, 10 6). Retrieved from axios: <https://axios-http.com/docs/intro>
- cloud.google*. (2012, 10 3). Retrieved from cloud.google: <https://cloud.google.com/docs>
- code.visualstudio*. (2018, 1 20). Retrieved from code.visualstudio:  
<https://code.visualstudio.com/docs>
- datascientest*. (2023, 05 16). Retrieved from Microsoft SQL Server : Tout ce qu'il faut savoir: <https://datascientest.com/microsoft-sql-server-tout-savoir>
- docs.djangoproject*. (2020, 06 22). Retrieved from djangoproject:  
<https://docs.djangoproject.com/en/5.0/>
- docs.github*. (2016, 10 6). Retrieved from docs.github: <https://docs.github.com/en>
- docs.stripe*. (2016, 10 02). Retrieved from docs.stripe: <https://docs.stripe.com/api>
- egacy.reactjs.org*. (2020, 02 02). Retrieved from legacy:  
<https://legacy.reactjs.org/docs/getting-started.html>
- https://restfulapi.net*. (2012, 02 11). Retrieved from <https://restfulapi.net>:  
<https://restfulapi.net>
- kinsta*. (2022, 12 19). Retrieved from kinsta: <https://kinsta.com/fr/base-de-connaissances/qu-est-express-js/>
- kinsta*. (2022, 12 19). Retrieved from Qu'est-ce qu'Express.js ? Tout ce que vous devez savoir: <https://kinsta.com/fr/base-de-connaissances/qu-est-express-js/>
- learning.postman.com*. (2020, 8 10). Retrieved from learning.postman.com:  
<https://learning.postman.com/docs/introduction/overview/>
- reactrouter*. (2020, 10 16). Retrieved from reactrouter: <https://reactrouter.com/en/main>
- redux*. (2015, 10 6). Retrieved from redux: <https://redux.js.org>
- sqlite*. (2006, 8 6). Retrieved from sqlite: <https://www.sqlite.org/docs.html>