# Dossier de Projet

## Projet Fil Rouge POEI 2025
## Restaurants Pâte d'Or

Du 16/12/2024 au 23/04/2025

Préparation au titre de Concepteur Développeur d'Applications

**Jean MUGNIERY**

inetum

m2i Formation

France Travail

adn ouest

# REMERCIEMENTS

Je tiens à remercier avant tout France Travail, ADN Grand Ouest et la coalition des entreprises (Inetum, CapGemini, Sogeti, Niji, Sigma) qui ont mis en place cette POEI. Je remercie tout particulièrement Inetum pour avoir cru en moi et m'avoir offert la chance de participer à cette formation.

Je remercie également M2I Formation pour l'encadrement de cette dernière et pour leur accueil chaleureux ces quatre mois. Je suis extrêmement reconnaissant envers Etienne Cassin, notre formateur principal, qui a réussi à faire de cette expérience intense un vrai plaisir, ainsi qu'à tous les autres formateurs qui sont intervenus auprès de nous.

Un grand merci aussi à Clara Laviale et Quentin Teillet avec qui j'ai travaillé sur le projet fil rouge. L'aide que nous nous sommes apportés pendant ce travail et la bonne ambiance qu'il y avait dans notre groupe a été très rafraichissante. Clara, merci de m'avoir cadré lorsque je partais dans tous les sens et notamment lorsque je fuyais Bootstrap. Quentin, merci pour le super boulot que tu as fait sur le côté graphique, mais aussi pour la gestion des réservations qui n'était pas une mince affaire.

Finalement, des remerciements chaleureux à tous nos autres camarades de promotion (Abdel, Régis, Maud, Valentin, Thierry, Nadège et Mickael). Nous nous sommes tous serrés les coudes et la camaraderie qu'il y a eu tout au long de ces quatre mois a fait passer le temps tellement vite qu'on en redemanderait presque !

## TABLE DES MATIERES

# Table des matières

# INTRODUCTION

Après avoir travaillé cinq ans en tant qu'assistant de recherche en Mathématiques à l'Université Catholique de Louvain, j'ai décidé de changer de voie et de m'orienter vers le développement.

C'est un domaine que je connaissais un petit peu car j'avais étudié une peu d'informatique durant mes études universitaires, et le cadre logique de résolution de problèmes que le développement apporte me plaisait beaucoup.

Dans ce contexte, j'ai postulé pour une POEI Concepteur Développeur d'Applications mise en place par France Travail, ADN Grand Ouest, une coalition d'entreprises (Inetum, CapGemini, Sogeti, Niji, Sigma) et encadrée par M2I Formation.

Ce dossier de projet retrace le projet fil rouge que j'ai réalisé pendant la formation en équipe avec Clara Laviale et Quentin Teillet, deux camarades de promotion, sous la direction de Etienne Cassin notre formateur principal.

Il illustre une grande partie (mais pas toutes !) des compétences que j'ai acquises durant ces quatre mois intenses d'apprentissage et de mise en pratique.

# 1 Contexte du projet

Dans le cadre de notre POEI Concepteur Développeur d'Applications chez M2I Formation, nous avons dû réaliser un projet de gestion de restaurants. Celui-ci est décomposé en trois applications :

- Une application d'administration réservée au propriétaire de la chaine. Elle lui permet de gérer ses restaurants : ajout, modification, suppression d'un restaurant ; gestion des cartes et des plats... Cette dernière est une application Java SE avec une interface console simple.
- Une application pour le public, permettant aux clients de consulter les restaurants en ligne, voir les menus et effectuer des réservations. Celle-ci est une application JEE (avec Hibernate), avec une partie front end pour le site vitrine réalisée en HTML / CSS / Javascript. Nous utiliserons aussi Bootstrap pour rendre l'application plus facilement responsive.
- Une application pour les équipes de services, permettant la gestion des réservations ainsi que les prises de commandes et leur suivi. Il s'agit d'une application réalisée avec Java Spring pour le back end et Angular pour le front end.

Ce projet a été réalisé en équipe avec Clara LAVIALE et Quentin TEILLET qui suivent la même formation et sous la direction de notre formateur principal Etienne CASSIN.

# 2 Outils et gestion de projet

Nous avons commencé par définir les outils dont nous aurons besoin pour réaliser les différentes parties de notre projet.

- Pour le développement Java, nous avons choisi d'utiliser Eclipse et IntelliJ Community Edition. Nous avons installé la version Java Developer & Web Developer car nous allons utiliser (entre autres) JEE
- SQL Server et Microsoft SQL Server Management Studio pour la gestion de notre base de données relationnelles
- VSCode pour la partie front end (HTML / CSS / Javascript) car c'est un IDE performant et modulable grâce aux différentes extensions que l'on peut ajouter
- Git pour le versioning ainsi que GitHub pour pouvoir travailler de manière collaborative au long du projet
- Jira pour la gestion de projet

Pour la gestion de projet, nous avons décidé d'appliquer des méthodes Agiles, notamment SCRUM. Au vu de l'organisation de la formation et des applications, nous avons décidé de nous organiser en trois grands sprints, un par application. Nous avons réparti les tâches en un certain nombre de tickets correspondant à des User Stories mises en place avec notre formateur qui jouait le rôle du client. Nous avons regroupé ces dernières en plusieurs Epics, une par application.

Au début de chaque sprint, nous avons estimé la complexité des tâches à l'aide d'un poker planning, ainsi que leur valeur afin de décider lesquelles prioritiser. A chaque sprint, nous avons créé un tableau Kanban dans lequel nous avons traité les tickets du backlog de sprint. Pour qu'un ticket soit considéré validé, il faut l'accord des deux autres membres du projet.

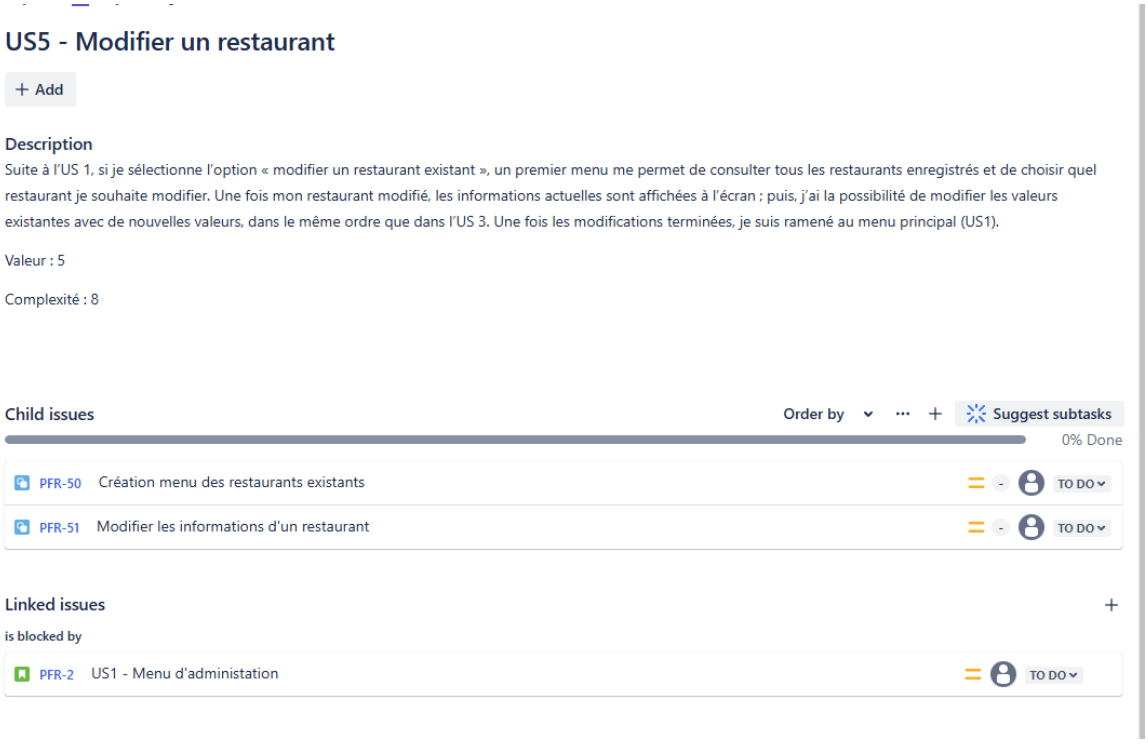


*Figure 1: Exemple de User Story*

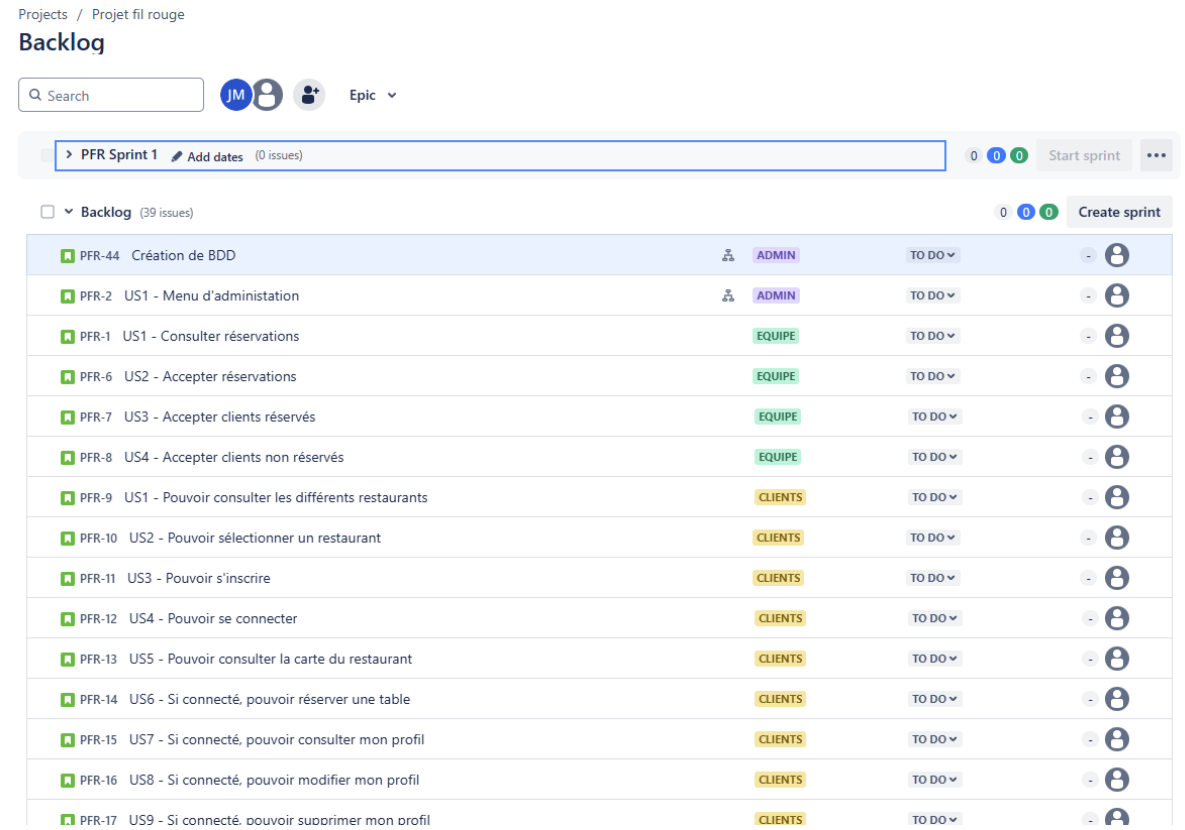


*Figure 2: Backlog avec Epics*

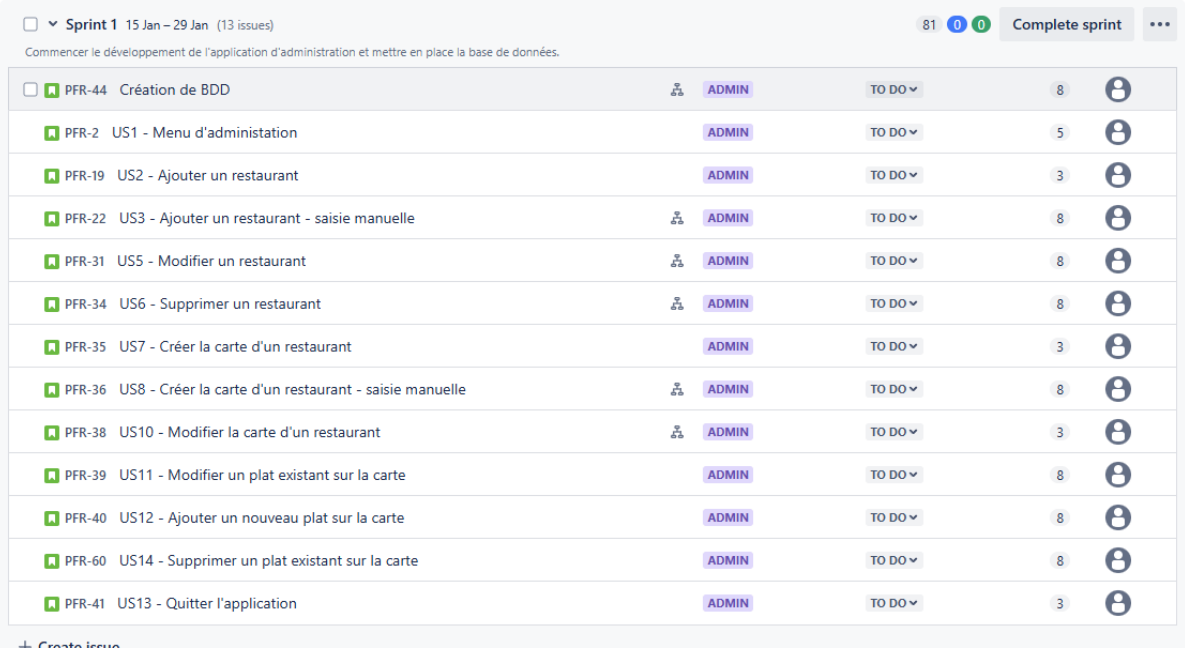


*Figure 3: Notre premier sprint*

Pour le travail collaboratif avec Github, nous avons choisi de travailler sur une branche develop à partir de laquelle chacun tire une branche correspondant au ticket qu'il a choisi de traiter. Pour pouvoir merge avec la branche develop, nous avons décidé que tous les membre du projet devaient valider la pull request. Cela permet à chacun de voir le code des autres et de pouvoir quand même se former sur des points qu'on n'aurait pas nous même développé.

Lors de la réalisation de la première application, nous avons créé la base de données relationnelle qui va nous servir tout au long du projet. Nous avons pour le début du projet, choisi d'avoir chacun une base de données personnelle en local, tout en mettant en commun les scripts de génération de base ainsi que les données de test. Nous avons aussi utilisé des variables d'environnement pour pouvoir accéder à nos bases de données respectives sans avoir à changer le code.

Pour l'application grand public, nous avons décidé d'installer les plugins suivant pour VSCode

- Live Server pour pouvoir lancer un serveur en local et voir les modifications qu'on effectue en temps réel
- VS Code Runner pour pouvoir exécuter du code sans avoir à passer par le terminal.

Comme nous avions de plus en plus de dépendances à gérer pour Java, nous avons aussi créé un projet Maven pour que la gestion de ces dernières se fasse de manière plus facile. Nous avons aussi choisi d'utiliser l'ORM Hibernate pour simplifier nos requêtes Java auprès de la base de données.

Pour l'application des équipes nous avons séparé l'application en deux. La partie backend a été réalisée avec Spring Boot et nous avons choisi Angular pour l'application frontend.
Nous avons utilisé Spring Initializr pour mettre le projet en place. Nous avons choisi d'utiliser Maven pour notre gestion de dépendances. Nous avons mis en place les dépendances :

- Lombok : pour faciliter la gestion de nos entités
- Spring Web : pour implémenter nos webservices
- Spring Data JPA : pour avoir accès à l'ORM Hibernate
- Spring Security : pour gérer la sécurité de notre application
- Des dépendances spécifiques à la gestion des tokens JWT

Nous avons aussi utilisé Postman pour tester nos webservices. J'ai créé un compte pour pouvoir sauvegarder une collection de requêtes de test pour chacun de nos endpoints. Cela nous aide lors du développement et nous permet de vérifier qu'il n'y a pas de régression au fur et à mesure que nous travaillons.

Pour la partie frontend, nous avons utilisé Figma pour la mise en place de maquettes. L'application a été réalisée avec Angular et Bootstrap. Nous avons développé avec Visual Studio Code.

# 3 Liste des compétences du référentiel qui sont couvertes par le(s) projet(s)

## 3.1 Application d'administration

Cette application fait intervenir les compétences suivantes :

- CP1 : Installer et configurer son environnement de travail en fonction du projet
- CP3 : Développer des composants métier
- CP4 : Contribuer à la gestion d'un projet informatique
- CP7 : Concevoir et mettre en place une base de données relationnelle
- CP8 : Développer des composants d'accès aux données SQL et NoSQL

## 3.2 Application grand public

Cette application fait intervenir les compétences suivantes :

- CP1 : Installer et configurer son environnement de travail en fonction du projet
- CP2 : Développer des interfaces utilisateurs
- CP3 : Développer des composants métier
- CP4 : Contribuer à la gestion d'un projet informatique
- CP5 : Analyser les besoins et maquetter une application
- CP6 : Définir l'architecture logicielle d'une application
- CP7 : Concevoir et mettre en place une base de données relationnelle
- CP8 : Développer des composants d'accès aux données SQL et NoSQL
- CP9 : Préparer et exécuter les plans de tests d'une application

## 3.3 Application équipes

Cette application fait intervenir les compétences suivantes :

- CP1 : Installer et configurer son environnement de travail en fonction du projet
- CP2 : Développer des interfaces utilisateurs
- CP3 : Développer des composants métier
- CP4 : Contribuer à la gestion d'un projet informatique
- CP5 : Analyser les besoins et maquetter une application
- CP6 : Définir l'architecture logicielle d'une application
- CP7 : Concevoir et mettre en place une base de données relationnelle
- CP8 : Développer des composants d'accès aux données SQL et NoSQL
- CP9 : Préparer et exécuter les plans de tests d'une application

# 4 Résumé du projet

## 4.1 Application d'administration

### Expression du besoin :

L'administrateur doit pouvoir, depuis la console d'administration :

- Ajouter un restaurant
- Modifier un restaurant existant
- Supprimer un restaurant existant
- Créer une carte
- Ajouter un plat à une carte
- Supprimer un plat d'une carte
- Modifier un plat d'une carte

### Gestion du projet :

La gestion de ce projet s'est faite à l'aide de méthodes agiles, notamment SCRUM. Nous avions une liste de user stories données par notre formateur qui jouait le rôle du client. Nous avons alors commencé par les étoffer pour déterminer ce qu'elles représentaient en termes de code, ainsi que leur valeur. Puis nous avons utilisé un poker planning pour évaluer la complexité de chaque US. Une fois ce travail fait, nous avons pu déterminer quelles US étaient prioritaires. Nous avons ensuite créé des tickets correspondants aux US sur Jira que nous avons ajouté au backlog et attribués entre nous. Une fois ce travail effectué, nous avons démarré notre sprint.

### Spécifications techniques :

L'application est développée avec Java SE. J'ai utilisé pour cela l'IDE IntelliJ Community Edition, avec un JDK Java 17.
Les données sont persistées dans une base de données SQL Server, accessible via Microsoft SQL Server Management Studio.

### Réalisation :

#### Mise en place de la base de données :

Pour la persistance des données, nous avons utilisé une base de données SQL Server que nous gérons avec Microsoft SQL Server Management Studio. Nous avons débattu de la structure de celle-ci avant le début du projet et avons décidé des différentes tables et colonnes. Nous avons mis en place les clés étrangères et les tables d'association reliant nos tables. Nous avons aussi généré un jeu de données de test qui nous servira tout au long du projet.

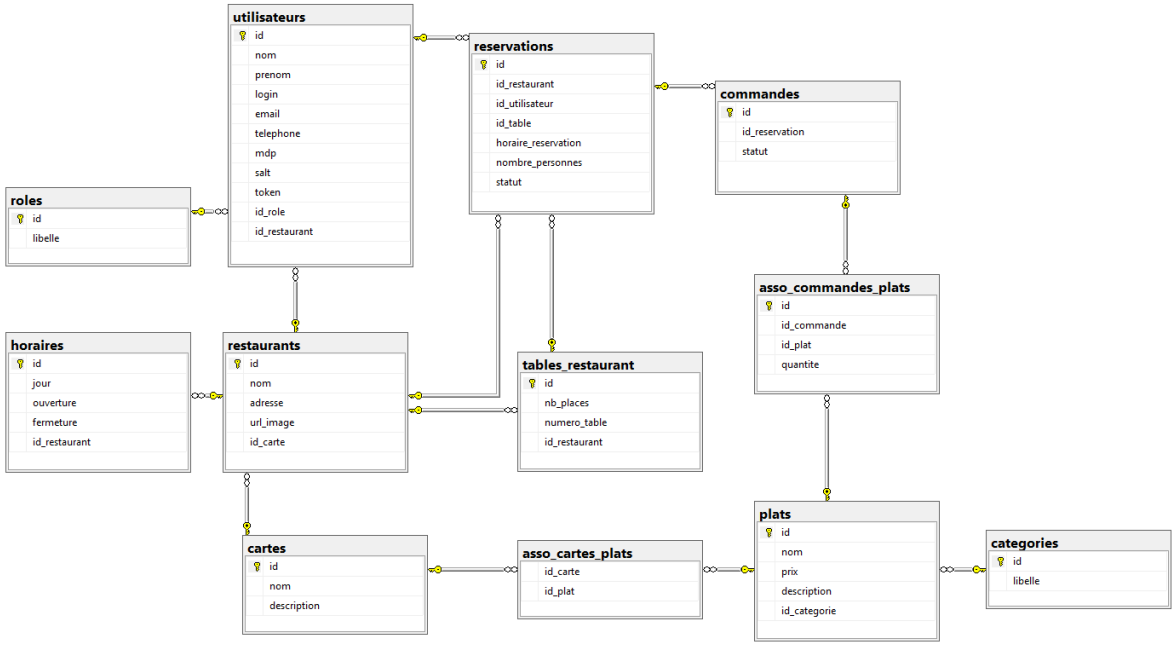


*Figure 4: Schéma de notre base de données*

```sql
DROP TABLE IF EXISTS    asso_commandes_plats,
                        asso_cartes_plats,
                        plats,
                        categories,
                        commandes,
                        reservations,
                        utilisateurs,
                        roles,
                        tables_restaurant,
                        horaires,
                        restaurants,
                        cartes;

CREATE TABLE cartes (
    id              INT                 PRIMARY KEY
                                        IDENTITY,
    nom             VARCHAR(30)         NOT NULL,
    description     VARCHAR(255)
);

CREATE TABLE restaurants (
    id              INT                 PRIMARY KEY
                                        IDENTITY,
    nom             VARCHAR(30)         NOT NULL,
    adresse         VARCHAR(255)        NOT NULL,
    url_image       VARCHAR(255)                ,
    id_carte        INT                 FOREIGN KEY REFERENCES cartes(id) ON DELETE SET NULL
);


CREATE TABLE horaires (
    id              INT                 PRIMARY KEY
                                        IDENTITY,
    jour            VARCHAR(8)          NOT NULL,
    ouverture       TIME                NOT NULL,
    fermeture       TIME                NOT NULL,
    id_restaurant   INT                 NOT NULL
                                        FOREIGN KEY REFERENCES restaurants(id) ON DELETE CASCADE
);

CREATE TABLE tables_restaurant (
    id              INT                 PRIMARY KEY
                                        IDENTITY,
    nb_places       INT                 NOT NULL,
    numero_table    INT                 NOT NULL,
    id_restaurant   INT                 NOT NULL
                                        FOREIGN KEY REFERENCES restaurants(id) ON DELETE CASCADE
);

CREATE TABLE roles (
    id              CHAR(3)             PRIMARY KEY,
    libelle         VARCHAR(30)         NOT NULL
                                        UNIQUE
);
```

*Figure 5:Script de création de tables*

InsertionDonneesTe...PFR [redacted](55))

```sql
INSERT INTO cartes (nom, description)
VALUES
('Carte Printemps', 'Carte de saison avec des plats frais printaniers'),
('Carte Été', 'Carte d'été avec des produits de saison'),
('Carte Hiver', 'Carte d'hiver avec des plats réconfortants'),
('Carte Automne', 'Carte automnale avec des recettes de saison'),
('Carte Végétarienne', 'Carte exclusivement végétarienne'),
('Carte Sans Gluten', 'Carte sans gluten'),
('Carte Desserts', 'Carte dédiée aux desserts gourmands'),
('Carte Boissons', 'Carte des boissons et cocktails');

INSERT INTO restaurants (nom, adresse, url_image, id_carte)
VALUES
('Le Gourmet', '10 Rue de la Gastronomie, Paris', 'https://images.unsplash.com/photo-1544148103-0773bf10d330', 1),
('Chez Mamie', '15 Rue des Amis, Lyon', 'https://images.unsplash.com/photo-1549488344-1f9b8d2bd1f3', 2),
('La Table d''Or', '25 Boulevard du Sud, Marseille', 'https://images.unsplash.com/photo-1555396273-367ea4eb4db5', 3),
('Bistro Rive Gauche', '5 Place des Champs, Paris', 'https://images.unsplash.com/photo-1485182708500-e8f1f318ba72', 4),
('L''Épicurienne', '30 Avenue du Ciel, Bordeaux', 'https://images.unsplash.com/photo-1600093463592-8e36ae95ef56', 5),
('Le Petit Coin', '40 Rue des Vins, Nantes', 'https://images.unsplash.com/photo-1554118811-1e0d58224f24', 6);

INSERT INTO horaires (jour, ouverture, fermeture, id_restaurant)
VALUES
-- Horaires pour Le Gourmet (id_restaurant = 1)
('Mardi', '10:00', '22:00', 1),
('Mercredi', '10:00', '22:00', 1),
('Jeudi', '10:00', '22:00', 1),
('Vendredi', '10:00', '23:00', 1),
('Samedi', '10:00', '23:00', 1),
('Dimanche', '10:00', '22:00', 1),

-- Horaires pour Chez Mamie (id_restaurant = 2)
('Lundi', '10:00', '22:00', 2),
('Mardi', '10:00', '22:00', 2),
('Mercredi', '10:00', '22:00', 2),
('Jeudi', '10:00', '22:00', 2),
('Vendredi', '10:00', '23:00', 2),
('Samedi', '10:00', '23:00', 2),

-- Horaires pour La Table d'Or (id_restaurant = 3)
('Lundi', '10:00', '22:00', 3),
('Mardi', '10:00', '22:00', 3),
('Jeudi', '10:00', '22:00', 3),
('Vendredi', '10:00', '23:00', 3),
('Samedi', '10:00', '23:00', 3),
('Dimanche', '10:00', '22:00', 3),

-- Horaires pour Bistro Rive Gauche (id_restaurant = 4)
('Lundi', '10:00', '22:00', 4),
('Mercredi', '10:00', '22:00', 4),
('Jeudi', '10:00', '22:00', 4),
('Vendredi', '10:00', '23:00', 4),
('Samedi', '10:00', '23:00', 4),
('Dimanche', '10:00', '22:00', 4),
```

*Figure 6: Script d'insertion des données*

Pour cette première application, nous avons juste besoin d'interagir avec les tables Restaurants, Tables, Plats, Cartes, et Horaires, mais nous avons tout de même mis en place l'intégralité de la base de données. Cela nous a permis d'avoir un jeu de données commun et complet pour l'intégralité du projet, tout en étant conscients que ce script serait amené à évoluer à mesure du développement de nos applications

Architecture de l'application :

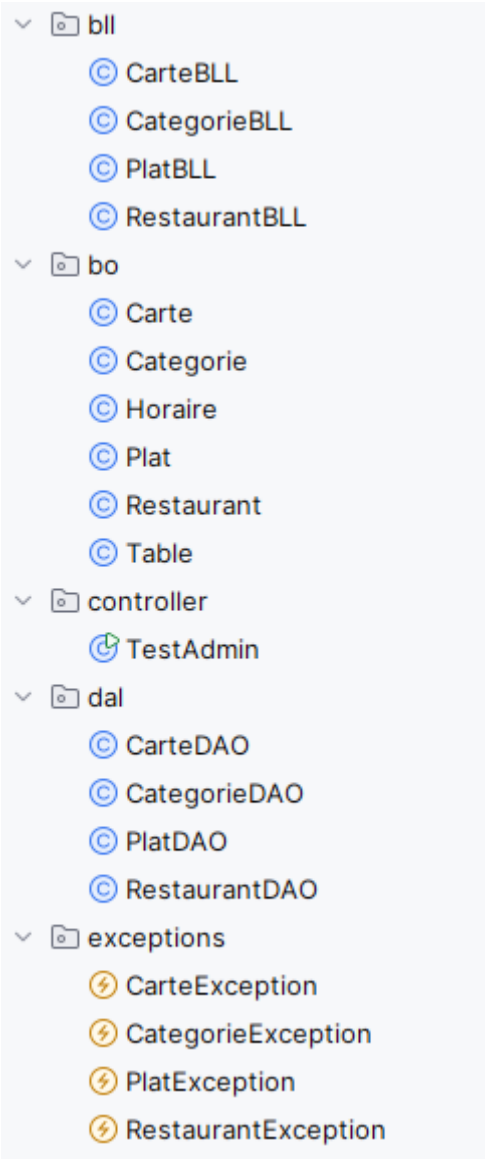


*Figure 7: Architecture de l'application d'administration*

Nous avons utilisé pour ce projet le design pattern DAO. Nous avons créé trois packages dans l'IDE : bo pour les objets métiers, bll pour la logique métier et dal pout l'accès à la base de données. Pour cette application le controller est assez simple : il est simplement responsable de naviguer entre les différents sous-menus qui apparaissent dans la console.

Les objets métiers nécessaires pour cette application sont représentés par les classes Restaurant, Carte, Plat, Horaire, Table. Ce sont des POJOs (Plain Old Java Object) qui ne contiennent donc que les attributs qui correspondent aux colonnes de notre base de données, ou dont les classes responsables du traitement ont besoin. Ils ne possèdent pas de méthodes autres que les différents constructeurs, getters et setters.

```java
public class Carte {  38 usages  Purukogi +1

    private int id;  4 usages
    private String nom;  6 usages
    private String description;  6 usages
    private String nomRestaurant;  3 usages
    private List<Plat> plats = new ArrayList<>();  5 usages

    public Carte(int id, String nom, String description, List<Plat> plats) {...}

    public Carte(String nom, String description, List<Plat> plats) {...}

    public Carte(String nom, String description) {...}

    public Carte() {  2 usages  Purukogi
    }

    public void ajouterPlat(Plat plat) { plats.add(plat); }

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    public String getNom() { return nom; }

    public void setNom(String nom) { this.nom = nom; }

    public String getDescription() { return description; }
```

*Figure 8: Exemple d'objet du BO*

Pour chacun de ces objets, nous avons ensuite créé une classe correspondante dans le package bll (p.e. Restaurant et RestaurantBLL). Dans cette classe, nous avons implémenté des méthodes qui vérifient la validité des données des objets du bo avant de les insérer en base de données. Puisque pour cette application, nous avons juste besoin des méthodes du CRUD, nous avons simplement écrit une méthode de vérification que nous appelons avant les insertions ou les updates. Celle-ci lance une exception spécifique à l'objet métier en question, dont le message explicite le problème.

```java
public Restaurant insert(String nom, String adresse, String url_image, Carte carte) throws RestaurantException {

        Restaurant restaurant = new Restaurant(nom, adresse, url_image, carte);
        checkRestaurantInsert(restaurant);
        dao.insert(restaurant);
        return restaurant;
}

public void update(Restaurant restaurant) throws RestaurantException {  1 usage   Clara
    checkRestaurantUpdate(restaurant);

    dao.update(restaurant);
}

private void checkRestaurantInsert(Restaurant restaurant) throws RestaurantException {  1 usage   Clara
    if (restaurant.getNom() == null || restaurant.getNom().isBlank()) {
        throw new RestaurantException("Le nom du restaurant doit être renseigné. Entrée pour continuer");
    }

    if (restaurant.getAdresse() == null || restaurant.getAdresse().isBlank() ) {
        throw new RestaurantException("L'adresse du restaurant doit être renseignée. Entrée pour continuer");
    }

    if (restaurant.getNom().length() > 30) {
        throw new RestaurantException("Le nom doit faire au maximum 30 caractères. Entrée pour continuer");
    }

    if (restaurant.getAdresse().length() > 255) {
        throw new RestaurantException("L'adresse doit faire au maximum 255 caractères. Entrée pour continuer");
    }

    if (restaurant.getUrl_image() != null && restaurant.getUrl_image().length() > 255) {
        throw new RestaurantException("L'URL de l'image doit faire au maximum 255 caractères. Entrée pour continue
    }
}
```

*Figure 9: Méthodes de la classe RestaurantBLL*

Finalement, dans le package dal, nous avons ajouté une classe correspondante à celle du bo (p.e. RestaurantDAO), responsable uniquement de traiter les requêtes SQL nécessaires. Si aucune exception n'est lancée dans la classe bll, alors la classe du dal appelle la méthode correspondante qui interagira avec la base de données. Nous assurons ainsi la sécurité de cette dernière en empêchant toute donnée incompatible d'être persistée.

Pour communiquer avec la base de données au travers des classes de la couche dal, nous avons utilisé JDBC pour effectuer des requêtes en SQL natif. Nous avons mis en place des variables d'environnement sur chacun de nos postes pour pouvoir nous connecter à la base de données, sans faire apparaitre les identifiants de connexion ou le nom de la base de données dans le code que l'on push sur GitHub. Nous avons implémenté les méthodes du CRUD ainsi que certaines contenant des requêtes plus complexes pour des besoins plus spécifiques.

```
public class RestaurantDAO {  3 usages   Clara +1 *

    String url = System.getenv( name: "FIL_ROUGE_URL");  4 usages
    String username = System.getenv( name: "FIL_ROUGE_USERNAME");  4 usages
    String password = System.getenv( name: "FIL_ROUGE_PASSWORD");  4 usages


    public List<Restaurant> select() {  1 usage   Clara +1 *
        List<Restaurant> restaurants = new ArrayList<>();

        try {
            Connection cnx = DriverManager.getConnection( url: "jdbc:sqlserver://"
                    + url
                    +";databasename=PFR;username="
                    + username
                    + ";password="
                    + password
                    + ";trustservercertificate=true");

            if(!cnx.isClosed()) {

                PreparedStatement ps = cnx.prepareStatement( sql: "SELECT r.id, r.nom, r.adresse, r.url_image, r.id_carte," +
                        " ca.nom AS carte_nom, ca.description AS carte_description"
                         + " FROM restaurants r"
                         + " LEFT JOIN cartes ca ON r.id_carte = ca.id;");
                ResultSet rs = ps.executeQuery();

                while (rs.next()) {
                    restaurants.add(convertResultSetToRestaurant(rs));
                }
            }
            cnx.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return restaurants;
    }
```

*Figure 10: Début de la classe RestaurantDAO*

```java
public List<Plat> select(int idCarte) {  1 usage   Purukogi +1 *

    List<Plat> plats = new ArrayList<>();

    try {
        Connection cnx = DriverManager.getConnection( url: "jdbc:sqlserver://"
                + url
                + ";databasename=PFR;username="
                + username
                + ";password="
                + password
                + ";trustservercertificate=true");

        if(!cnx.isClosed()){
            PreparedStatement ps = cnx.prepareStatement( sql: "SELECT p.id, p.nom, p.prix, p.description, " +
                    "c.libelle AS categorie_libelle " +
                    "FROM plats p " +
                    "INNER JOIN categories c ON p.id_categorie = c.id " +
                    "INNER JOIN asso_cartes_plats acp ON p.id = acp.id_plat " +
                    "INNER JOIN cartes ca ON acp.id_carte = ca.id " +
                    "WHERE ca.id = ?");
            ps.setInt( parameterIndex: 1, idCarte);
            ResultSet rs = ps.executeQuery();
            while(rs.next()){
                plats.add(convertResultSetToPlat(rs));
            }
        }
        cnx.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return plats;
}
```

*Figure 11: Récupération des plats d'une carte*

La requête ci-dessus permet de récupérer tous les plats d'une carte, ainsi que le libellé de leur catégorie. Pour cela nous avons dû passer par la table association_cartes_plats et la table categories à l'aide d'INNER JOIN.

L'IHM est simplement une liste de menus et de saisies utilisateurs dans la console, donc nous traitons tout dans une classe Main. Cette classe est responsable de l'IHM, ce dernier consistant en une liste de menus et de saisies utilisateurs dans la console. Nous avons donc décidé d'implémenter une méthode responsable de l'affichage de chaque sous-menu.

Notons que nous avons utilisé la classe PreparedStatement pour effectuer nos requêtes SQL. Elle effectue des contrôles sur les paramètres que l'on met dans nos requêtes pour empêcher les injections SQL en échappant certains caractères que l'on retrouve en SQL.

Lorsque l'utilisateur interagit avec la base de données (p.e. quand il ajoute un restaurant, ou modifie une carte), la classe du bll correspondante est appelée, puis si les données entrées sont cohérentes, alors la classe du dal associée les ajoute en base.

## Réalisation personnelle :

Pour cette application, j'étais responsable de la partie création de carte et d'ajout de plats à une carte. J'ai commencé pour cela par créer la classe correspondante dans le BO, en lui ajoutant les constructeurs, getters et setters. Je me suis ensuite attelé à rédiger le début de l'interface utilisateur pour la gestion de cartes. Nous avions structuré le code de sorte que chaque sous-menu corresponde à une méthode propre, c'est donc dans celle-ci que j'ai travaillé.

### Ajout de carte :

J'ai utilisé un Scanner pour les saisies de l'utilisateur. Après avoir entré le nom et la description de la carte, j'ai voulu ajouter la carte à la base de données. J'ai donc créé les classes CarteBLL et CarteDAO pour l'insertion, ainsi que les méthodes correspondantes. J'ai aussi mis en place une méthode checkCarte dans le BLL pour m'assurer que la carte créée respecte les contraintes de notre base de données.

Une fois cela fait, j'ai ajouté un Try-Catch autour de l'insertion de la carte dans la classe Main, puis dans le cas d'une insertion réussie, j'ai écrit une boucle Do-While pour que l'utilisateur puisse ajouter autant de plats qu'il le souhaite. Pour cela j'ai créé une méthode saisiePlat() pour le faire saisir les informations relatives aux plats, créé les classes PlatBLL et PlatDAO ainsi que les méthodes d'insertion en base et de vérification des contraintes. Une fois le plat saisi, j'ai encore une fois utilisé un Try-Catch pour enrober l'insertion. Puisqu'un plat peut être associé à plusieurs cartes, j'ai aussi dû écrire une méthode associerPlatCarte() qui remplit la ligne correspondante dans la table Asso_cartes_plats.

```java
private static void sousMenuCreationCarte() {  1 usage   Purukogi +1
    String nom;
    String description;
    int choix;

    System.out.println("Nom de la carte :");
    nom = scan.nextLine();
    System.out.println("Description de la carte :");
    description = scan.nextLine();

    CarteBLL bll = new CarteBLL();
    try {
        Carte carte = bll.insert(nom, description);
        System.out.println("Carte créée avec succès !");
        do{
            System.out.println("Voulez-vous ajouter un plat à la carte ?");
            System.out.format(" %-7s %s\n", "1.", "Oui\n");
            System.out.format(" %-7s %s\n", "2.", "Non\n");
            try{
                choix = scan.nextInt();
                if (choix == 2){
                    continue;
                }
            } catch (InputMismatchException e) {
                System.err.println("Choix invalide.");
                choix = -1;
            } finally {
                scan.nextLine();
            }


             Plat plat = saisiePlat();
             carte.ajouterPlat(plat);
             try {
                 PlatBLL platBLL = new PlatBLL();
                 platBLL.insert(plat);
                 platBLL.associerPlatCarte(plat, carte);
             } catch (PlatException e){
                 System.err.println("Erreur lors de la création du plat : " + e.getMessage());
             }
             System.out.println("Plat ajouté avec succès.");

        }while(choix != 2);
        System.out.println("les plats ont tous bien été ajoutés");
    } catch (CarteException e) {
        System.out.println("Erreur lors de la création de la carte : " + e.getMessage());
    }
}
```

*Figure 12: Méthode de création de carte*

Ajout de plat à une carte :

Les méthodes de création, d'insertion, de vérification et d'association ayant déjà été créées, l'ajout de plat a été simple à implémenter. J'ai juste rassemblé toutes ces méthodes dans celle correspondant à l'ajout de plat à une carte. A noter que l'id de la carte a dû être traqué pour savoir à laquelle ajouter le plat.

### Conclusion :

Arrivés à la fin de notre sprint, nous avons fait un sprint review. Nous n'avons pas réussi à développer toutes les fonctionnalités prévues dans le backlog. En effet il nous manquait la gestion des horaires, ainsi que la possibilité de supprimer et de modifier des plats. Nous avons donc décidé de reporter ces tickets au sprint suivant.

Nous avons constaté que certains de nos tickets n'avaient pas été assez étoffés, ce qui a fait que nous sommes passés à côté de certaines choses. De plus, lors de l'attribution des tickets, nous n'avons pas bien réfléchi aux dépendances de nos tâches, ce qui a fait que nous nous bloquions mutuellement.

Cette partie du projet, bien que relativement simple techniquement, nous a permis de relever plusieurs points d'améliorations, notamment sur la gestion de projet. Cependant, c'était notre premier sprint et nous avons conscience que les méthodes agiles fonctionnent de manière itérative, donc nous avons mis en place des plans d'amélioration pour le sprint suivant : une meilleure préparation préliminaire des tickets, ainsi qu'une répartition des tâches plus minutieuses, quitte à passer plus de temps lors du sprint planning.

## 4.2 Application grand public

### Expression du besoin :

Cette application consiste en un site vitrine destiné au public, regroupant les informations des différents restaurants (adresse, horaire, menu) et doit permettre à un client d'effectuer une réservation ainsi que de contacter un restaurant. L'application doit pouvoir :

- Permettre au client d'accéder à la page de chaque restaurant
- Permettre au client de consulter la carte et les horaires d'un restaurant
- Permettre au client de se créer un compte ainsi que de se connecter
- Si le client est connecté, lui permettre de faire une demande de réservation
- Si le client est connecté, lui permettre de contacter un restaurant
- L'application doit être accessible à la fois sur ordinateur et sur mobile

### Gestion de projet :

Nous avons procédé comme pour l'application précédente. Nous avons revu les user stories une par une lors d'un sprint planning afin de déterminer non seulement leur valeur et

complexité, mais aussi leur implémentation (JSP, Servlets...). Une fois ce travail effectué, nous avons réparties les tickets entre nous selon les fonctionnalités.

## Spécifications techniques :

La base de données de cette application est partagée avec celle de la première.
Le backend de cette application a été réalisé en Java JEE à l'aide de l'IDE Eclipse. Nous avons utilisé un serveur Tomcat 11, ce qui, pour des raisons de compatibilité, nous force à utiliser un JDK 17. Nous avons aussi utilisé Maven pour la gestion de dépendances. Nous avons utilisé l'ORM Hibernate pour la communication avec la base de données.
Le frontend, quant à lui, a été réalisé en partie sur VSCode avec le module Live Server pour le HTML / CSS / Javascript et dans Eclipse pour les spécificités liées aux JSPs. Nous avons aussi utilisé le framework Bootstrap pour rendre le site responsive. Nous avons réalisé les maquettes du site avec wireframes.cc.

## Réalisation du projet :

Nous avons commencé par créer des maquettes pour le site, ainsi qu'un schéma de navigation (*voir plus loin*) pour s'assurer que nous ayons une bonne vision des JSPs, des Servlets qui les lient et des méthodes à appeler. Une fois ce travail effectué, nous avons mis en place nos packages. Nous avons, comme pour l'application d'administration, appliqué le design pattern DAO, que nous avons complété avec le pattern MVC.

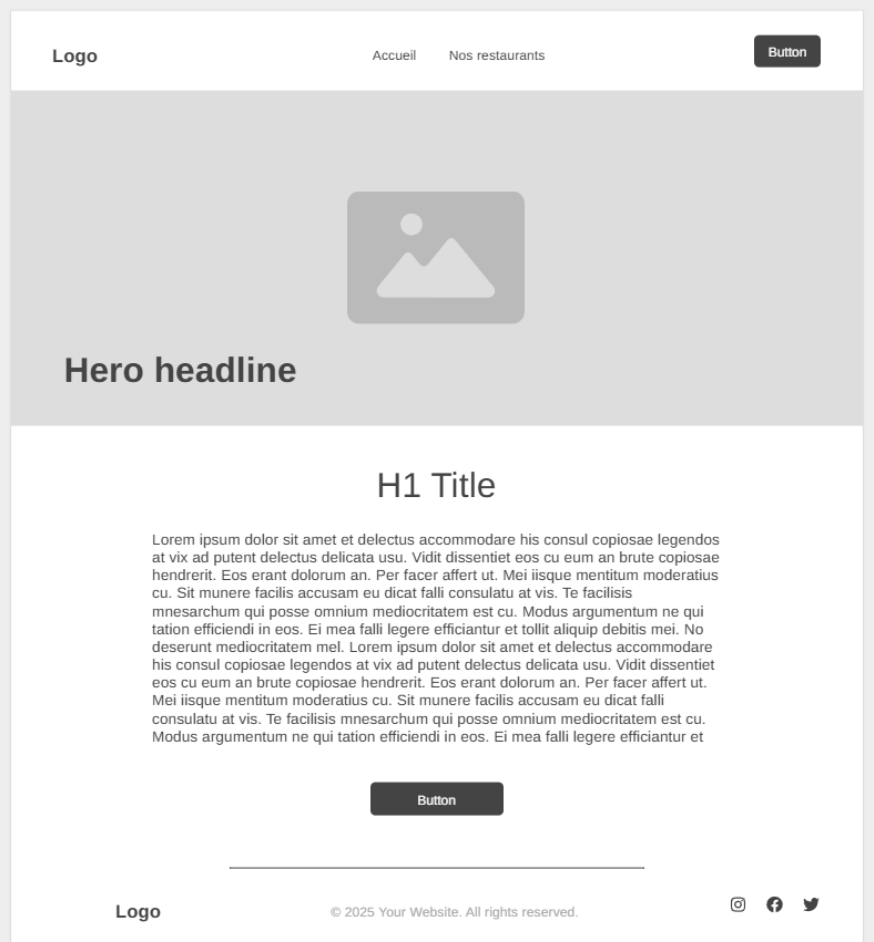


*Figure 13: Maquette de la page d'accueil*

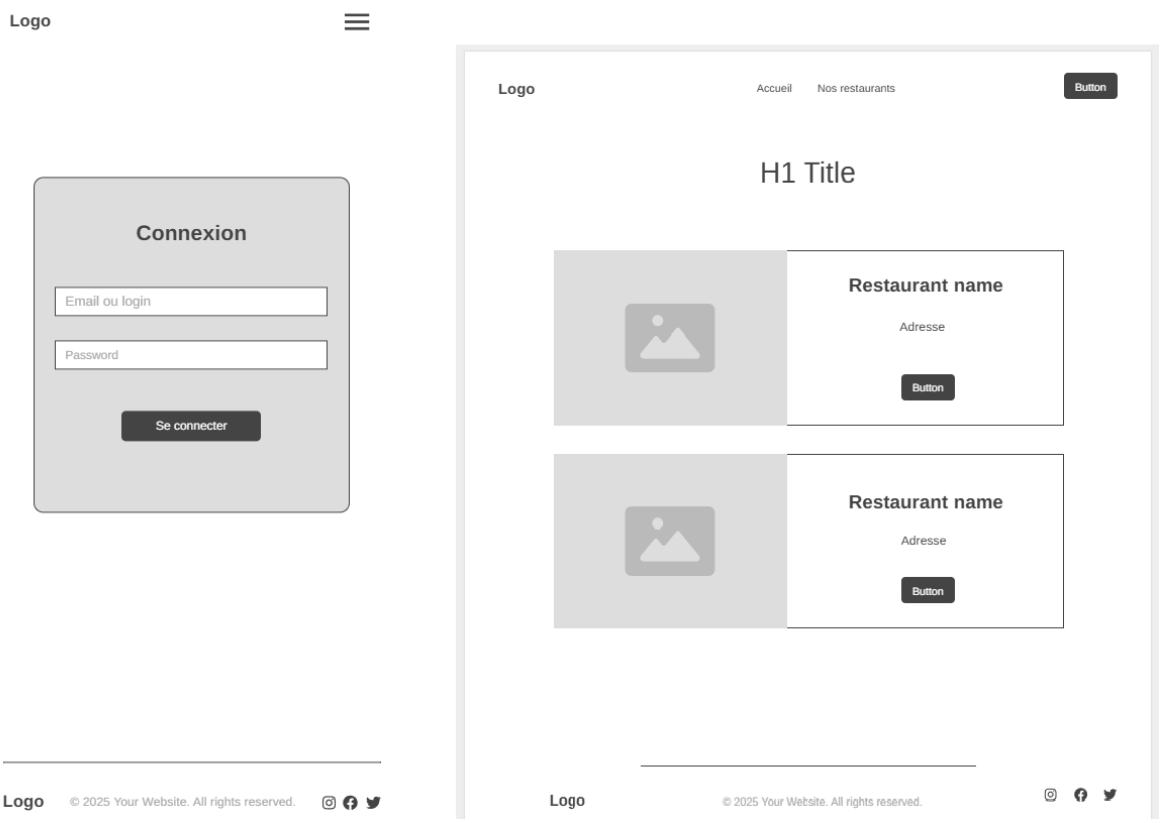


*Figure 14: Maquette du formulaire de connexion mobile et de la page listant les restaurants*

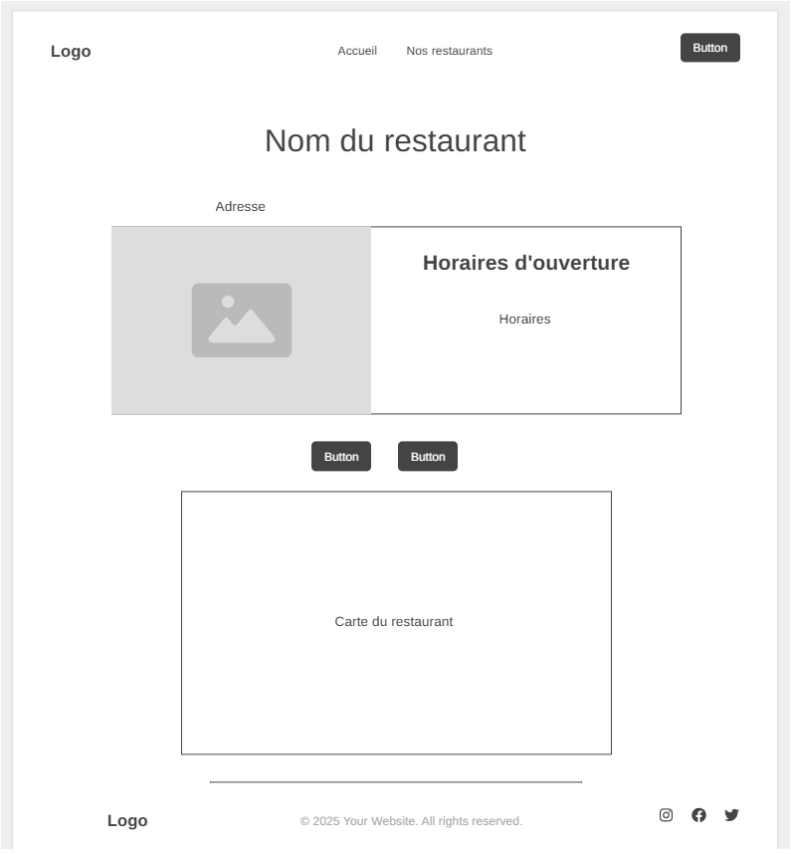


*Figure 15: Maquette de la page d'un restaurant*

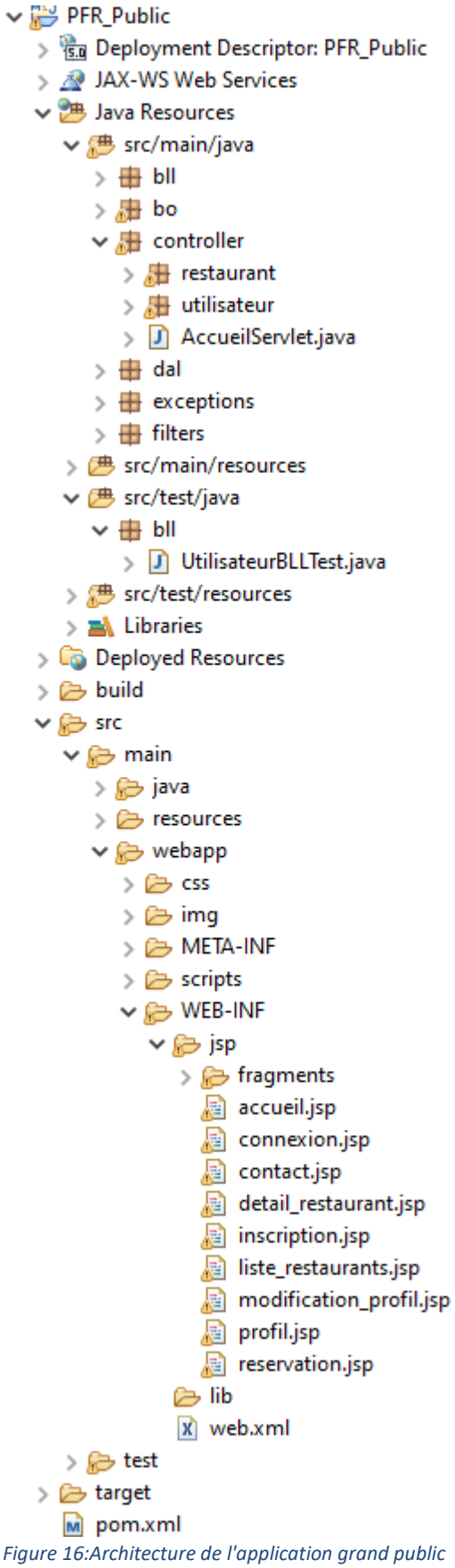


*Figure 16:Architecture de l'application grand public*

Nous avons donc créé un projet Maven, dans lequel nous avons ajouté les packages bo, bll, dal et exceptions avec les mêmes rôles que pour la première application. Nous les avons complétés avec un package controller contenant nos Servlets, ainsi qu'un package filters contenant les filtres que nous allons mettre en place pour gérer les connexions. Nous avons mis les JSPs dans le dossier src/main/webapp/WEB-INF afin de s'assurer qu'elles ne soient pas accessibles sans passer par une Servlet.

Nous avons commencé par créer les classes du bo. Comme nous utilisons l'ORM Hibernate, nous avons dû réfléchir à la manière d'annoter les attributs de nos classes en fonction de nos dépendances. Nous avons utilisé les annotation @OneToOne, @OneToMany et @ManyToOne. Nous avons aussi dû nous assurer que les tables déjà dans notre base de données correspondaient à nos classes. Cela a été mis en place à l'aide de l'annotation @Table.

```java
@Entity @Table(name = "cartes")
public class Carte {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String nom;
    private String description;

    @ManyToMany
    @JoinTable(
            name = "asso_cartes_plats",
            joinColumns = {@JoinColumn(name = "id_carte")},
            inverseJoinColumns = {@JoinColumn(name = "id_plat")}
    )
    private List<Plat> plats;

    @Transient
    private Map<Categorie, List<Plat>> platsGroupedByCategory;

    public Carte(int id, String nom, String description, List<Plat> plats) {
        this.id = id;
        this.nom = nom;
        this.description = description;
        this.plats = plats;
    }

    public Carte(String nom, String description, List<Plat> plats) {
        this.nom = nom;
        this.description = description;
        this.plats = plats;
    }

    public Carte() {}
```

*Figure 17:Exemple d'entité*

Une fois ce travail effectué, nous avons pu commencer à avancer sur les JSPs et les Servlets. Dans le but d'éviter de la redondance dans le code, nous avons décidé d'écrire deux fragments pour le header et le footer que nous avons ensuite injecté sur toutes nos JSPs.

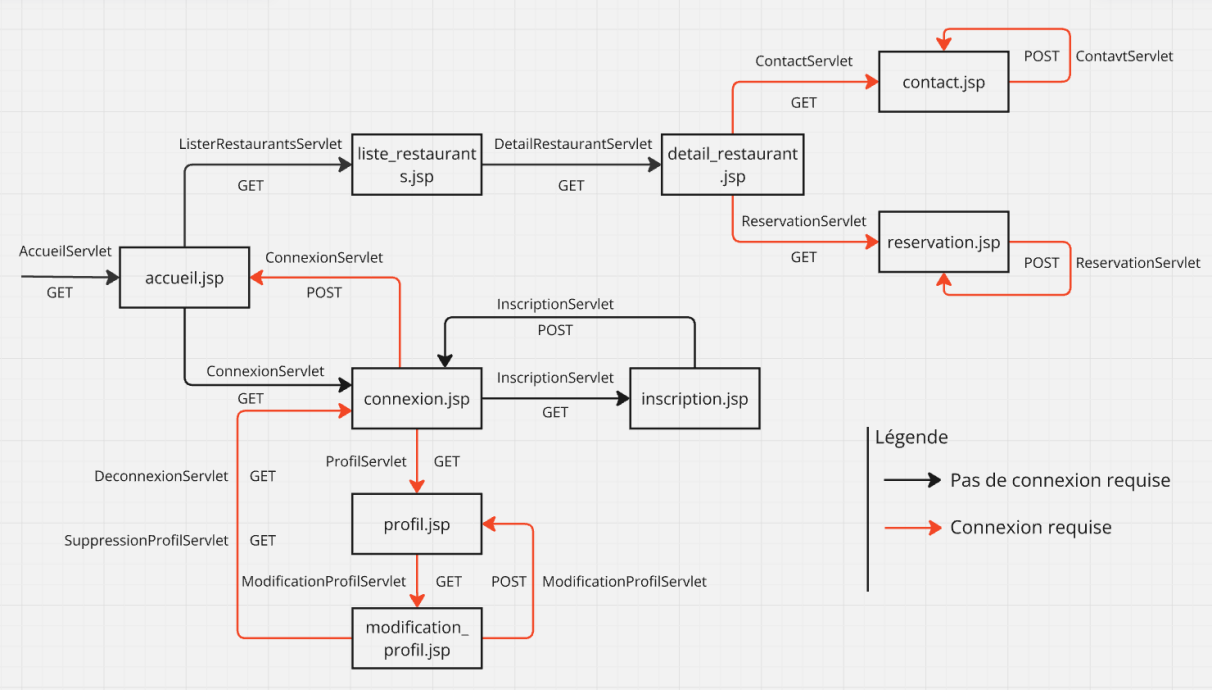


*Figure 18: Schéma de navigation*

Du schéma de navigation, on voit que :

- Nous avons choisi d'utiliser les méthodes GET de nos Servlets lorsque nous n'envoyons pas d'informations sensibles dans la requête, et POST lorsqu'on le fait. Cela évite par exemple que les informations de connexion apparaissent dans l'URL.
- Certaines actions requièrent une connexion. C'est naturel pour les pages relatives aux informations de profil, quant aux réservations et contacts, cela a été mis en place à la demande du client. Cela permettra, par exemple, par la suite d'implémenter un compte fidélité.

Pour mettre en place l'obligation de connexion, nous avons utilisé une classe filtre. Pour simplifier le schéma, la Servlet de connexion renvoie vers l'accueil, mais nous avons en fait mis en place une redirection vers la page précédente. En effet, le lien vers la page de connexion se trouvant dans le header (et donc sur toutes les pages), il est préférable pour l'expérience utilisateur que ce dernier ne soit pas systématiquement redirigé vers l'accueil.

```
@WebFilter(
        dispatcherTypes = DispatcherType.REQUEST,
        urlPatterns = {"/contact", "/reservation", "/profil", "/modification-profil"}
)
public class ConnexionFilter extends HttpFilter implements Filter {
    private static final long serialVersionUID = 1L;

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
    ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        Utilisateur utilisateur = (Utilisateur) httpRequest.getSession().getAttribute("utilisateur");
        if (utilisateur != null) {
            chain.doFilter(httpRequest, httpResponse);
            return;
        }

        String requestedUrl = httpRequest.getRequestURI();
        String queryString = httpRequest.getQueryString();
        if (queryString != null) {
            requestedUrl += "?" + queryString;
        }
        httpRequest.getSession().setAttribute("redirectAfterLogin", requestedUrl);

        httpResponse.sendRedirect("connexion");
    }
}
```

*Figure 19: Filtre de connexion*

Nous avons aussi mis en place un système de token, afin d'implémenter une manière pour l'utilisateur de rester connecté. Un token lui est attribué lors de la connexion et stocké en base de données. Lorsque l'utilisateur visite le site, un filtre vérifie si un token correspondant à l'un de la base de données est présent dans les cookies. Si c'est le cas, l'utilisateur est automatiquement connecté.

Pour rendre le site dynamique, nous récupérons les informations nécessaires à la page demandée en base de données, puis nous les incluons dans la requête. Pour les afficher, nous utilisons de l'Expression Language dans nos JSPs. Dans le cas où nous avons, par exemple, une liste à afficher, nous avons utilisé les librairies JSTL.

```java
@WebServlet("/lister-restaurants")
public class ListerRestaurantsServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static RestaurantBLL restaurantBLL = new RestaurantBLL();

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
    IOException {
        List<Restaurant> restaurants = restaurantBLL.select();
        request.setAttribute("listeRestaurants", restaurants);
        request.getRequestDispatcher("/WEB-INF/jsp/liste_restaurants.jsp").forward(request, response);
    }

}
```

*Figure 20:Envoi d'information au travers des requêtes http*

```jsp
<section class="taille-navbar container d-flex flex-column align-items-center justify-content-center">
<h2 class="mb-5 mt-5">Nos restaurants</h2>

<div class="row justify-content-center" >
<c:forEach items="${listeRestaurants}" var="r" varStatus="bStatus">
    <fieldset class="mb-4 col-lg-8 col-md-12 text-center">
        <div class="row border border-2 rounded align-items-center" >
            <div class="col-md-6 m-0 p-0 " style="max-height: 250px; overflow: hidden;">
                <img src="${r.url_image}" alt="Image de ${r.nom}" class="img-fluid restaurant-image" style="max-height: 250px;">
            </div>
            <div class="col-md-6 pt-2 pb-2 d-flex flex-column justify-content-between">
                <h4 >${r.nom}</h4>
                <p>${r.adresse}</p>
                <form action="detail-restaurant" method="GET">
                    <input type="hidden" name="id" value="${r.id}">
                    <input type="submit" value="Plus d'informations" class="btn btn-primary">
                </form>
            </div>
        </div>
    </fieldset>
</c:forEach>
</div>
```

*Figure 21: Affichage dynamique de l'information avec expression language et JSTL*

Une fois les fonctionnalités mises en place, nous nous sommes attelés à respecter les maquettes. Nous avons pour cela utilisé le framework CSS Bootstrap. Comme nous voulions avoir un site responsive, nous nous sommes assurés d'éviter d'utiliser des tailles fixes et de préférer du display flex.

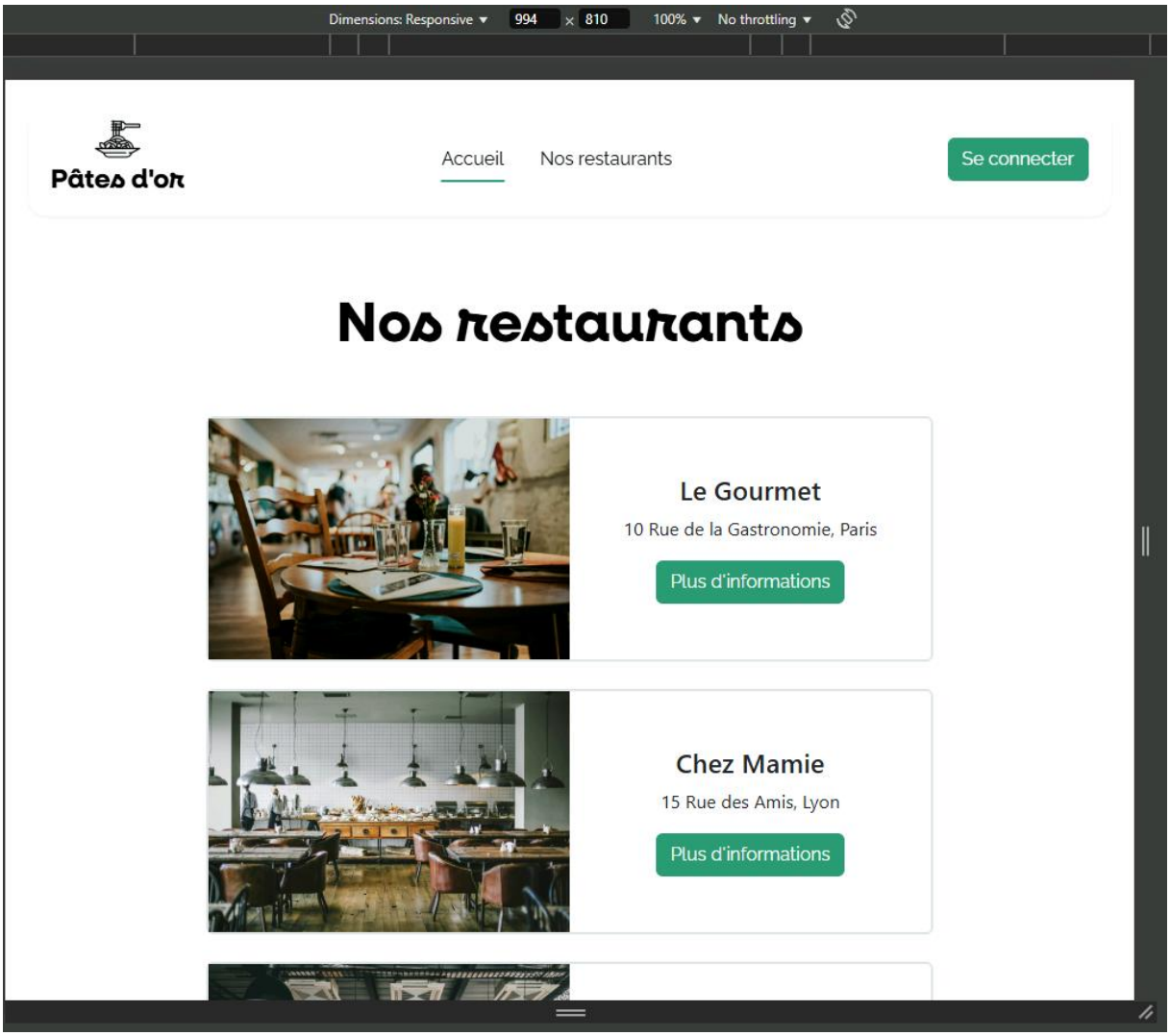


*Figure 22:Page listant les restaurants sur écran réduit*

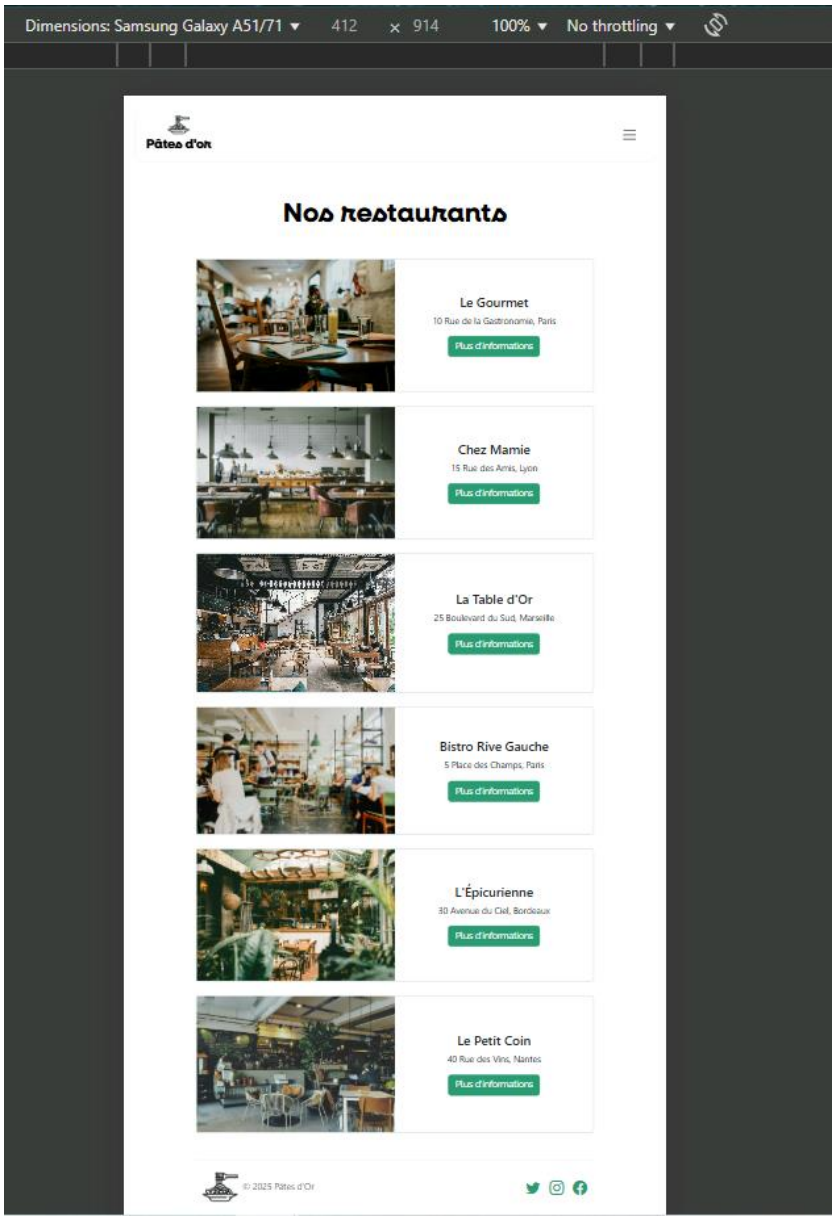


*Figure 23: Page listant les restaurants sur mobile*

## Réalisation personnelle :

Pour cette application, j'étais chargé de plusieurs fonctionnalités :

- Le formulaire et le service d'inscription
- Le formulaire et le service de connexion
- La page profil et certains des services qui en découlent (modification, suppression, déconnexion)
- Les Servlets et front des pages ci-dessus
- La sécurisation des informations de connexion
- Le service de token pour rester connecté
- La mise en place de tests unitaires dans Eclipse

J'ai commencé par mettre en place le service de connexion, afin de pouvoir générer des mots de passe cryptés pour nos jeux de test. J'ai commencé par créer le formulaire d'inscription, qui appelle la Servlet InscriptionServlet en POST. Avant d'insérer l'utilisateur en base de données plusieurs méthodes de la classe UtilisateurBLL sont appelées pour s'assurer de la sécurité des information saisies. Tout d'abord une méthode checkUtilisateur() vérifie que les informations sont bien conformes aux contraintes métier et de la base de données (*p.e.* login, mdp non nuls, longueur des champs) ainsi qu'une méthode vérifiant que l'email est bien valide à l'aide d'une Regex.

```
public void checkUtilisateur(Utilisateur client) throws UtilisateurException {

    UtilisateurException exception = new UtilisateurException();

    if (client.getNom() == null || client.getNom().isBlank()) {
        exception.addMessage("Le nom ne peut pas être laissé vide !");
    }

    if (client.getPrenom() == null || client.getPrenom().isBlank()) {
        exception.addMessage("Le prénom ne peut pas être laissé vide !");
    }

    if (client.getEmail() == null || client.getEmail().isBlank()) {
        exception.addMessage("L'e-mail ne peut pas être laissé vide !");
    }

    if (!(client.getNom() == null) && client.getNom().length() > 30) {
        exception.addMessage("Le nom ne peut pas faire plus de 30 caractères !");
    }

    if (!(client.getPrenom() == null) && client.getPrenom().length() > 30) {
        exception.addMessage("Le prénom ne peut pas faire plus de 30 caractères !");
    }

    if (!(client.getLogin() == null) && client.getLogin().length() > 30) {
        exception.addMessage("L'identifiant ne peut pas faire plus de 30 caractères !");
    }

    if (!(client.getEmail() == null) && client.getEmail().length() > 60) {
        exception.addMessage("L'e-mail ne peut pas faire plus de 60 caractères !");
    }

    if (!(client.getTelephone() == null) && client.getTelephone().length() > 20) {
        exception.addMessage("Le numéro de téléphone ne peut pas faire plus de 20 caractères !");
    }
```

```
    if(!(client.getEmail() == null) && !checkEmail(client.getEmail())) {
        exception.addMessage("L'adresse e-mail n'est pas valide !");
    }

    if (exception.getMessages().size() > 0) {
        throw exception;
    }

}

public boolean checkEmail(String email) {
    //^[A-Za-z0-9_-] begins with a block of any letter, number _ or -
    //+(\\.[A-Za-z0-9_-]+)* add any number of blocks ".String" (where String contains any letter, number, _ or -)
    //[^-] can't start after @ with -
    //[A-Za-z0-9-] first block is any letter, number or -
    //+(\\.[A-Za-z0-9-]+)* then we add any number of blocks ".String" (where String contains any letter, number or -)
    //(\\.[A-Za-z]{2,})$ then we end with a bloc ".String" (where String contains any letter and is at least 2
    //characters long)
    return Pattern.compile("^[A-Za-z0-9_-]+(\\.[A-Za-z0-9_-]+)*@[^-][A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$")
            .matcher(email)
            .matches();
}
```

*Figure 24: Vérification des informations du client*

Si un des champs n'est pas conforme, une exception est levée, dont la liste des messages contient les informations non conformes. Un encart s'affiche alors sur la page d'inscription pour laisser l'utilisateur quels champs posent problème.

Dans le cas où les informations sont correctes, alors le mot de passe est crypté par une série de méthodes du BLL. Nous avons choisi pour cela d'utiliser l'algorithme de cryptage PBKDF2. Pour cela, nous commençons par générer un salt : un tableau de bytes aléatoires qui sera gardé en base de données et qui servira à paramétrer la méthode de hashage du mot de passe. Une fois le salt généré, le mot de passe ainsi que ce dernier sont passés en paramètre de la méthode de hashage et le mot de passe crypté est stocké dans en base.

```
private void generateSalt(Utilisateur client) {

    SecureRandom random = new SecureRandom();
    byte[] salt = new byte[16];
    random.nextBytes(salt);
    client.setSalt(salt);
}

public byte[] hashMdp(String mdp, byte[] salt) {

    KeySpec spec = new PBEKeySpec(mdp.toCharArray(), salt, 65536, 128);
    try {
        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        byte[] hash = factory.generateSecret(spec).getEncoded();
        return hash;
    } catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
        e.printStackTrace();
    }
    return null;
}
```

*Figure 25: Génération du salt et hashage du mot de passe*

Pour le formulaire de connexion, j'ai commencé par créer la Servlet ConnexionServlet et écrit la méthode GET pour rediriger l'utilisateur vers la page voulue. Puis, pour coller à la maquette, j'ai créé une div avec la classe card en Bootstrap, dans laquelle j'ai inséré un formulaire appelant ConnexionServlet en POST.

La méthode correspondante dans la Servlet essaie de récupérer un utilisateur en fonction de son login ou de son adresse mail. Si un utilisateur est trouvé, alors le mot de passe envoyé par le formulaire est hashé, puis comparé au mot de passe hashé de l'utilisateur en base.

Si ceux-ci correspondent, alors l'utilisateur est connecté et stocké dans la session pour que les services nécessitant une connexion le sachent. Si le mot de passe ou l'identifiant ne correspondent pas à un utilisateur en base, une exception est lancée et la JSP affiche un message d'erreur.

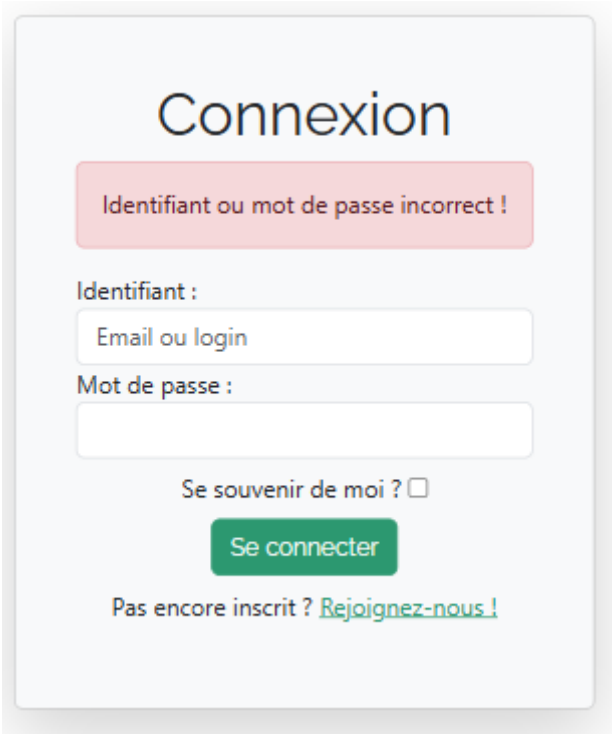


*Figure 26: Message d'erreur en cas de mauvais login*

Si l'utilisateur clique sur "Se souvenir de moi ?", alors un token est généré aléatoirement et ajouté à la ligne correspondant à l'utilisateur. Puis ce dernier est placé dans un cookie qui est donné à l'utilisateur. Quand l'utilisateur arrive sur une page du site, un filtre vérifie dans ses cookies s'ils contiennent un cookie appelé "token". Si c'est le cas, et que son contenu correspond au token d'un utilisateur en base, alors cet utilisateur est ajouté à la session et il est par conséquent considéré connecté.

```
@WebFilter(
        dispatcherTypes = DispatcherType.REQUEST,
        urlPatterns = "/*"
)
public class SouvenirFilter extends HttpFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
    ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        Cookie[] cookies = httpRequest.getCookies();
        if (cookies != null) {
            for (Cookie current : cookies) {
                if ("token".equals(current.getName())) {
                    UtilisateurBLL bll = new UtilisateurBLL();
                    Utilisateur userByToken = bll.selectByToken(current.getValue());
                    if (userByToken != null) {
                        httpRequest.getSession().setAttribute("utilisateur", userByToken);
                        chain.doFilter(httpRequest, httpResponse);
                        return;
                    }
                }
            }
        }
        chain.doFilter(httpRequest, httpResponse);
    }
}
```

*Figure 27: Connexion grâce au token stocké dans les cookies*

Pour la page profil, j'ai créé un formulaire contenant les informations de l'utilisateur stocké dans la session, mais dont les champs sont désactivés. Puis j'ai ajouté les boutons de modification, déconnexion et suppression de compte.

Le bouton de modification appelle ModificationProfilServlet en GET pour rediriger l'utilisateur vers une page contenant un formulaire avec les informations de l'utilisateur en session préremplies.

L'utilisateur peut modifier les champs qu'il désire, et envoyer les modifications en POST à ModificationProfilServlet. Plusieurs vérifications sont mises en place lors de l'update en base de données.

Si les informations ne respectent pas les contraintes métiers ou que le mot de passe entré est incorrect, une exception est levée et un message d'erreur s'affiche indiquant toutes les informations incorrectes. Les informations de l'utilisateur sont ré-insérées dans les champs du formulaire.

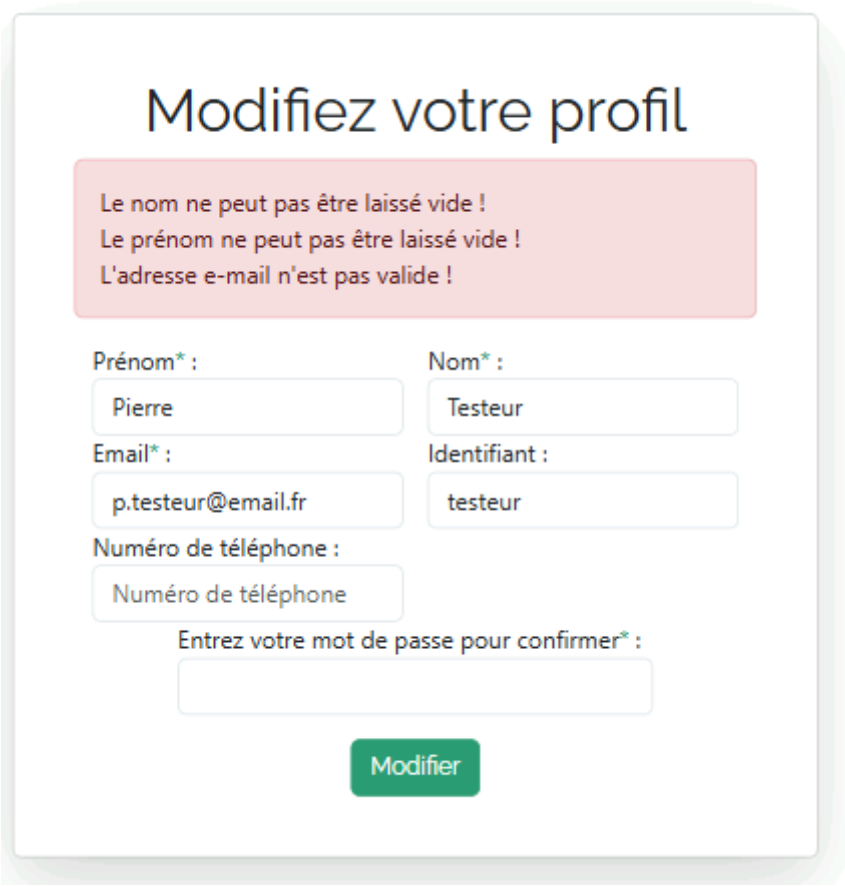


*Figure 28: Messages d'erreur en cas d'informations incorrectes*

Certaines des méthodes dont j'avais besoin pour mettre en place ces différentes fonctionnalités demandaient des requêtes SQL plus fines que celle du CRUD. J'ai donc mis en place des named querries pour pouvoir accéder aux données que je voulais récupérer.

```
@NamedQueries({
    @NamedQuery (name = "selectByToken",
                 query="SELECT u FROM Utilisateur u WHERE u.token= :token"),
    @NamedQuery (name = "selectByLogin",
                 query="SELECT u FROM Utilisateur u WHERE u.login= :login"),
    @NamedQuery (name = "selectByEmail",
                 query="SELECT u FROM Utilisateur u WHERE u.email= :email"),
    @NamedQuery (name = "selectByEmailEtMdp",
                 query="SELECT u FROM Utilisateur u WHERE u.email= :email AND u.mdp= :mdp"),
    @NamedQuery (name = "selectByLoginEtMdp",
                 query="SELECT u FROM Utilisateur u WHERE u.login= :login AND u.mdp= :mdp")
})
```

*Figure 29: Named querries pour récupérer des informations utilisateur*

La déconnexion est effectuée par la méthode GET de DeconnexionServlet. Elle se contente de supprimer l'instance d'utilisateur stocké en session ainsi que de supprimer le cookie qui contient le token de reconnexion s'il en existe un.Puis l'utilisateur est redirigé vers la page de connexion. Pour la suppression de compte, on appelle SuppressionProfilServlet en GET. Cette méthode supprime l'utilisateur en base de données, puis appelle DeconnexionServlet en GET.

Finalement, j'ai aussi mis en place des tests JUnits pour tester les méthodes du BLL. Dans le dossier source src/test/java, j'ai mis en place les classes de test. J'ai aussi créé une base de données de test que j'ai lié au projet à travers un fichier persistence.xlm que j'ai mis dans le dossier src/test/resources. Cela nous a permis de tester que les différents cas de succès ou d'échec des méthodes du BLL se comportent comme désiré.

```java
@BeforeAll
static void beforeAllInit() {
    bll = new UtilisateurBLL();
}

@Test
void checkUtilisateurValide_neFaitRien() {
    try {
        Utilisateur valide = new Utilisateur();
        valide.setPrenom("Etienne");
        valide.setNom("Cassin");
        valide.setEmail("e.cassin@email.fr");

        bll.checkUtilisateur(valide);
    } catch (UtilisateurException e){
        fail("L'utilisateur devrait etre validé.");
    }
}
```

*Figure 30: Initialisation de la classe à tester et premier exemple de test unitaire*

```java
@Test
void checkUtilisateurEmailInvalide2_renvoieUtilisateurException() {
    Utilisateur invalide = new Utilisateur();
    invalide.setPrenom("Etienne");
    invalide.setNom("Cassin");
    invalide.setEmail("e\".cassin@e-mail.co.uk");

    UtilisateurException e = assertThrows(UtilisateurException.class, () -> {
        bll.checkUtilisateur(invalide);
    });

    assertEquals(1, e.getMessages().size());
    assertEquals("L'adresse e-mail n'est pas valide !", e.getMessages().get(0));
}

@Test
void checkUtilisateurInvalideCombo_renvoieUtilisateurException() {
    Utilisateur invalide = new Utilisateur();
    invalide.setEmail("e-cassin@-email.com");

    UtilisateurException e = assertThrows(UtilisateurException.class, () -> {
        bll.checkUtilisateur(invalide);
    });

    assertEquals(3, e.getMessages().size());
    assertEquals("Le nom ne peut pas être laissé vide !", e.getMessages().get(0));
    assertEquals("Le prénom ne peut pas être laissé vide !", e.getMessages().get(1));
    assertEquals("L'adresse e-mail n'est pas valide !", e.getMessages().get(2));
}
```

*Figure 31: D'autres exemples de tests unitaires*

## Conclusion :

Cette application s'est beaucoup mieux déroulée que la première, notamment parce que nous avons mis en place plusieurs améliorations discutées lors de notre premier sprint review. En particulier nous avons passé beaucoup plus de temps lors de notre sprint planning pour étoffer nos différents tickets.

Comme cette application est réalisée en JEE, nous avons décidé, pour chaque ticket quelle(s) JSP(s) et quelle(s) Servlets sont nécessaires. Nous avons donc commencé par réaliser notre schéma de navigation à partir duquel nous nous sommes basés pour mettre nos objets en place. Nous nous sommes aussi réparti les tickets par thèmes, afin que les fonctionnalités que nous développions individuellement soient le plus indépendantes possibles.

J'ai trouvé que l'articulation Servlet / JSP était intéressante à mettre en place, car cela nous a forcé à mettre en place une certaine rigueur. De plus, cette manière de faire nous a permis de nous représenter plus facilement le schéma de navigation de notre application.

La partie connexion de l'application a été enrichissante. Réfléchir à la manière de stocker le mot de passe de l'utilisateur, à comment le crypter, et la gestion de l'option "se souvenir de moi" m'ont beaucoup appris. S'assurer de la qualité des données envoyées en base par nos formulaires à l'aide de contrôles par le backend et la mise en place des tests unitaires pour nous assurer du bon fonctionnement de ces derniers a été un bon défi à relever.

Enfin, mettre en place les écrans s'est avéré un challenge. Je n'avais pas beaucoup d'expérience avec le frontend et l'ajout de l'implémentation de Bootstrap a rendu l'apprentissage plus compliqué. Cependant grâce aux connaissances et à l'aide des membres de mon équipe, j'ai réussi à mettre en place des pages responsive. Mettre en place l'affichage des messages d'erreurs après les contrôles était intéressant aussi.

## 4.3 Application Equipes

### Expression du besoin :

Cette application est à usage des équipes, afin de gérer les réservations, l'arrivé des clients et la prise de commande. Elle sera séparée en deux parties : une application backend qui gère les accès à la base de données au travers de webservices et une application frontend avec laquelle les employés interagissent. L'application doit permettre aux employés :

- D'accepter ou refuser les demandes de réservation
- D'enregistrer les commandes des clients afin que celles-ci soient enregistrées en cuisine
- De générer les factures des clients

### Gestion de projet :

Nous avons procédé comme pour l'application précédente, à ceci près que nous avons mis en place un sprint par application (Backend et Frontend). Nous avons revu les user stories une par une lors d'un sprint planning afin de déterminer non seulement leur valeur et complexité, mais aussi pour les étoffer. Une fois ce travail effectué, nous avons réparties les tickets entre nous selon les fonctionnalités.

### 4.3.1 Application Backend

#### Spécifications techniques :

La base de données de cette application est partagée avec celle de la première.
Cette application est réalisée en Java avec le Framework Spring et l'IDE IntelliJ Community Edition. Pour la gestion de dépendances, nous avons choisi Maven et nous sommes passés par Spring Initializr. Nous avons mis en place les dépendances suivantes :

- Lombok : pour faciliter la gestion de nos entités
- Spring Web : pour implémenter nos webservices
- Spring Data JPA : pour avoir accès à l'ORM Hibernate
- Spring Security : pour gérer la sécurité de notre application
- Des dépendances spécifiques à la gestion des tokens JWT

Nous avons aussi utilisé Figma pour la réalisation des maquettes, ainsi que Postman pour tester nos webservices.

#### Réalisation du projet :

Cette application étant constituée uniquement de Webservices pour communiquer avec la partie frontend, nous avons commencé par créer des maquettes afin de savoir quels endpoints mettre en place et quelle quantité d'information nous avons besoin d'envoyer et de recevoir.

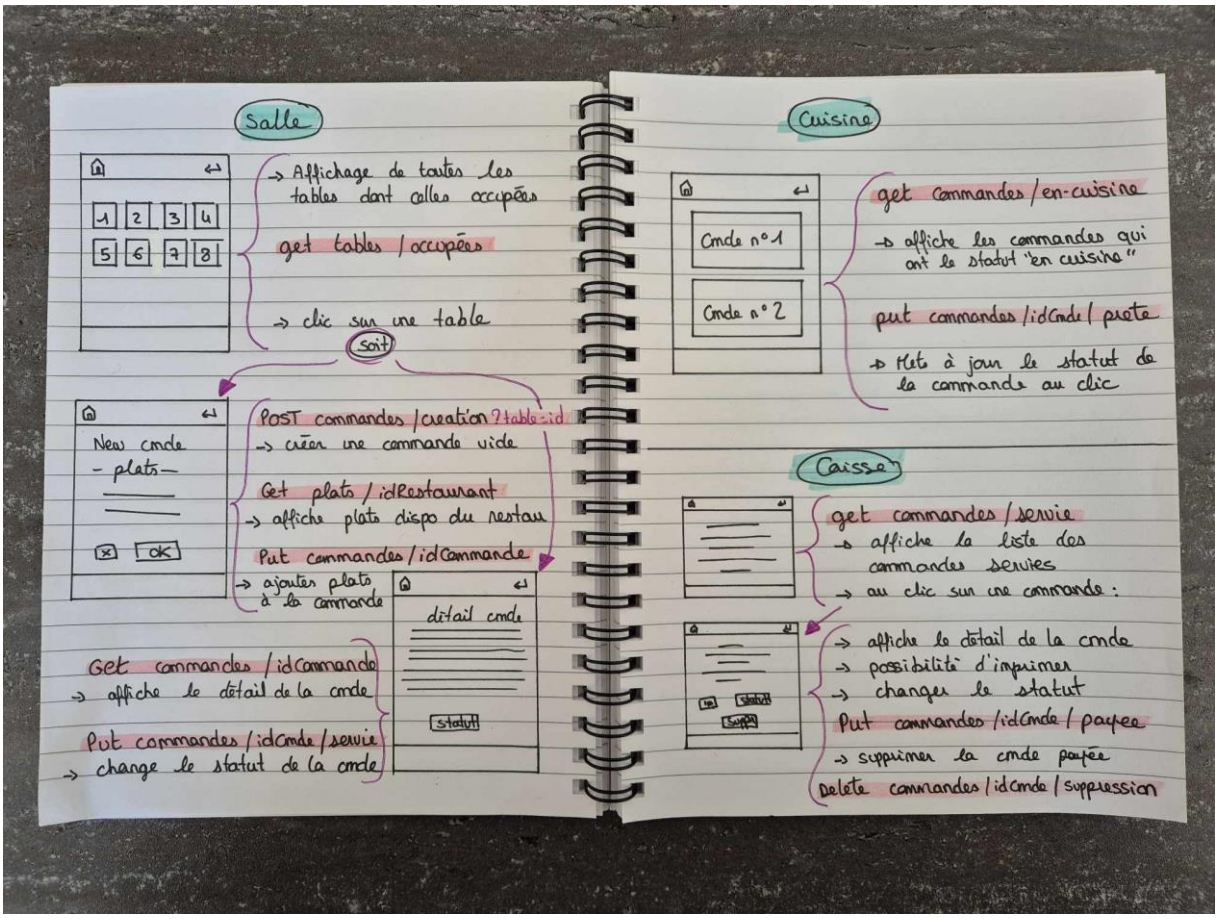


*Figure 32: Premier maquettage pour définir les endpoints de nos webservices*

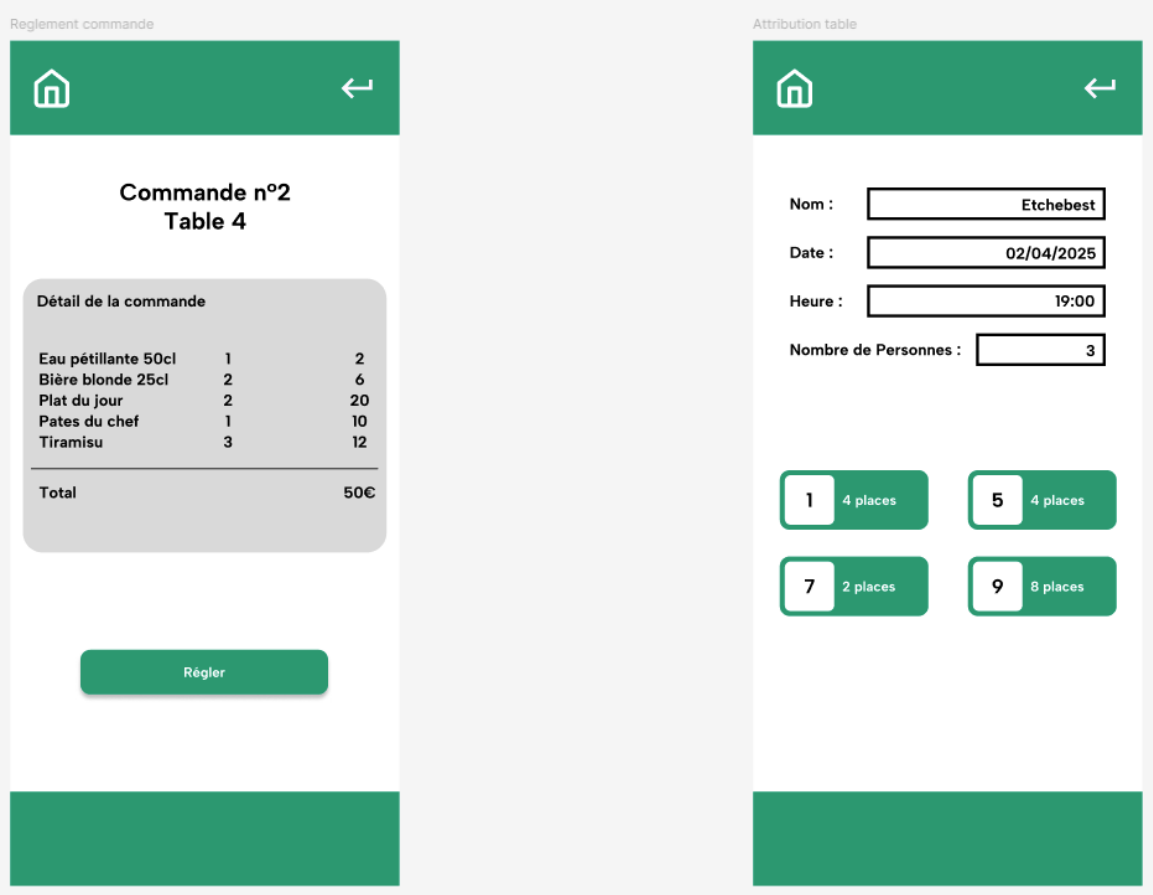


*Figure 33:Maquettes réalisées avec Figma*

Comme nous utilisons le framework Spring et que cette application est constituée uniquement de webservices, nous avons structuré le projet de la manière suivante. Nous avons mis en place trois packages pour gérer nos objets métiers :

- Un package entity qui contient les objets tels qu'ils sont persistés en base
- Un package dto qui contient les DTOs correspondants aux entités, afin de n'envoyer au front que les informations dont ce dernier a besoin
- Un package mapper qui contient les classes qui font la traduction des entités vers les DTOs et vice-versa

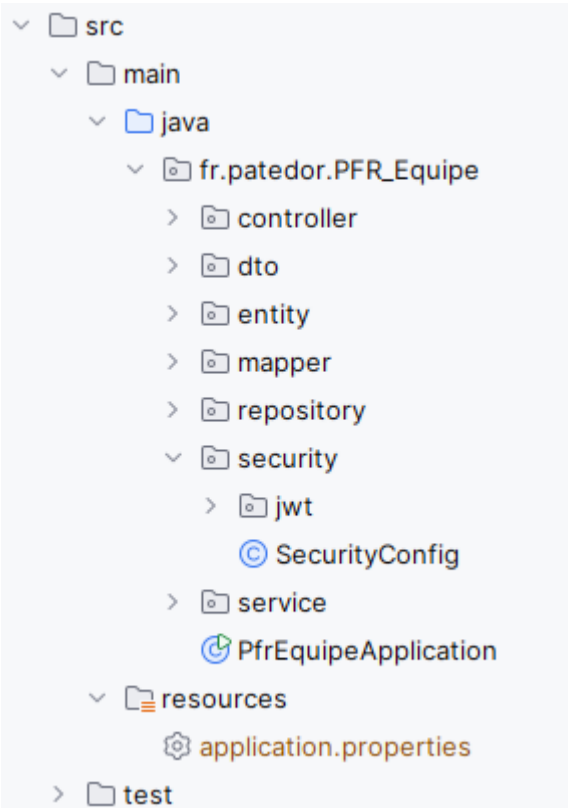


*Figure 34: Architecture du backend de l'application équipes*

Une fois la gestion des entités mise en place, nous avons mis en place nos Webservices. Pour cela nous avons créé un package controller dans lequel nous avons mis les classes responsables de nos différents endpoints, répartis par catégorie. Par exemple, la classe AdminController est responsable des endpoints pour les services d'administration : voir la liste des restaurants, la liste des employés, ajouter ou supprimer un employé...

Puis, nous avons créé deux packages : service et repository. Ils correspondent respectivement à nos couches BLL et DAO. Dans le package service, pour chaque classe, nous avons commencé par mettre en place un Interface pour chaque service et une implémentation de cet Interface. Cela nous permet, par exemple, de mettre en place des implémentations de test de nos services qui nous renvoient des informations données.

Le package repository quant à lui ne contient que des Interfaces, rendus très simples grâce au framework que nous utilisons. Nous avons, pour la plupart de nos requêtes, juste besoin de méthodes prédéfinies, ou définies par leur nom. Dans le cas où nous avons des requêtes plus fines à effectuer, nous avons utilisé Hibernate et du JPQL pour écrire ces dernières.

```
public interface TableRestaurantRepository extends JpaRepository<TableRestaurant, Integer> {  2 usages  Quentin +1

    @Query("SELECT t FROM TableRestaurant t WHERE t.restaurant.id = :restaurantId")  3 usages  Quentin
    List<TableRestaurant> findAllByRestaurantId(@Param("restaurantId") Integer restaurantId);

    @Query("SELECT t FROM TableRestaurant t WHERE t.numeroTable = :numeroTable AND t.restaurant.id = :idRestaurant")  1 usage  Quentin
    TableRestaurant findByNumeroTableAndIdRestaurant(@Param("numeroTable") Integer numeroTable, @Param("idRestaurant") Integer idRestaurant);

    @Query("SELECT t FROM TableRestaurant t WHERE t.restaurant.id = :idRestaurant AND t.nbPlaces + 1 >= :nbPersonne")  2 usages  Quentin
    List<TableRestaurant> findByNbPlacesAndIdRestaurant(@Param("idRestaurant") Integer idRestaurant, @Param("nbPersonne") Integer nbPersonne);

}
```

*Figure 35: Exemple de repository*

Finalement, nous avons mis en place un package security dans lequel sont stockées les classes relatives à notre implémentation de Spring Security pour gérer la sécurité et les connexions des utilisateurs. Le but étant de forcer une connexion pour que les employés puissent accéder à l'application, ainsi que de restreindre l'accès à certaines fonctionalités en fonction du rôle des employés (*p.e.* les fonctionnalités d'administrateur)

## Réalisation personnelle :

Pour cette partie de l'application j'étais responsables des tâches suivantes :

- L'écriture des webservices pour la partie administration
- La mise en place de Spring Security et des mécaniques de connexion
- Du renfort sur les webservices de gestion de réservation

Pour les webservices d'administration, j'avais quatre endpoints à mettre en place. J'ai commencé par créer une classe AdminController que j'ai annoté de @RestController et

@RequestMapping(“/admin”) pour que Spring sache que cette classe gère des webservices et que les urls de ceux-ci commencent tous par “/admin”.

Puis j’ai écrit les méthodes suivantes :

- GET : récupère la liste de tous les restaurants afin que l’administrateur puisse accéder à la liste des employés de chaque enseigne pour les gérer
- GET “/{id_restaurant}” : récupère la liste des employés du restaurant dont l’id est passé dans l’url
- POST + PUT “/{id_restaurant}” : ajoute et modifie respectivement un employé du restaurant passé en paramètre dans le corps de la requête
- DELETE “/{id_restaturant}/{id_employe}” : supprime l’employé dont l’id est passé dans l’url

```
@GetMapping  no usages  Purukogi
public ResponseEntity<List<RestaurantDTO>> getAll() {
    List<RestaurantDTO> restaurants = restaurantService.findAll().stream() Stream<Restaurant>
            .map( Restaurant restaurant -> restaurantMapper.toDTO(restaurant)) Stream<RestaurantDTO>
            .collect(Collectors.toList());
    return  ResponseEntity.ok(restaurants);
}
```

```
@GetMapping("/{id_restaurant}")  no usages  Purukogi
public ResponseEntity<List<EmployeDTO>> getEmployes(@PathVariable("id_restaurant") Integer idRestaurant) {
    List<EmployeDTO> employes = employeService.findFromRestaurant(idRestaurant).stream() Stream<Employe>
            .map( Employe utilisateur -> employeMapper.toDTO(utilisateur)) Stream<EmployeDTO>
            .collect(Collectors.toList());
    return ResponseEntity.ok(employes);
}
```

```
@PostMapping("/{id_restaurant}")  no usages  Purukogi *
public ResponseEntity<EmployeDTO> addEmploye(@RequestBody EmployeDTO utilisateur,
                                             @PathVariable("id_restaurant") Integer idRestaurant) {
    Optional<Restaurant> restaurant = restaurantService.findById(idRestaurant);
    Employe aAjouter = Employe.builder()
            .nom(utilisateur.getNom())
            .prenom(utilisateur.getPrenom())
            .login(utilisateur.getLogin())
            .email(utilisateur.getEmail())
            .telephone(utilisateur.getTelephone())
            .mdp(passwordEncoder.encode(utilisateur.getLogin().toLowerCase()))
            .role(new Role( id: "EMP",  libelle: "Employé"))
            .build();
    restaurant.ifPresent(aAjouter::setRestaurant);
    employeService.addEmploye(aAjouter);
    return ResponseEntity.ok(employeMapper.toDTO(aAjouter));
}
```

*Figure 36: Exemples de webservices admin*

Pour pouvoir mettre ces méthodes en place, j'ai eu besoin d'appeler plusieurs autres classes (*p.e.* RestaurantService, RestaurantMapper, PasswordEncoder...). Pour cela, j'ai laissé Spring gérer les injections de dépendances en ajoutant l'annotation @Autowired sur chacun d'entre eux.

J'ai ensuite écrit les méthodes dont j'avais besoin dans les classes correspondantes du package service (RestaurantService, EmployeService et leurs implémentations). Comme ce ne sont que des méthodes de CRUD, l'écriture n'a pas nécessité d'outils particuliers, les méthodes des classes de services se contentent de déléguer aux repositories correspondant qui ont aussi été annotés avec @Autowire.

Pour l'écriture des repositories, cela a été très rapide, puisque la majorité du travail est faite par Spring. Je me suis contenté de les faire étendre la classe JpaRepository<T, U> et d'écrire les requêtes plus fines que findAll, save ou delete (*p.e.* j'ai utilisé un @Query("FROM Employe e WHERE e.restaurant.id = :id") pour récupérer tous les employés d'un restaurant donné).

J'ai ensuite mis en place Spring Security sur le projet. Nous avons décidé de le laisser sur une branche à part jusqu'à la fin du développement, afin de pouvoir réaliser nos tests sur Postman sans avoir à gérer les étapes supplémentaires de connexion et de tokens JWT. Je faisais en revanche des merges réguliers de develop vers cette branche pour pouvoir tester nos endpoints.

Dans le but d'être le plus complet dans ces tests, j'ai créé une collection de requêtes sur Postman correspondant à tous les endpoints définis dans nos maquettes. Ces dernières sont testées régulièrement au fur et à mesure du développement.

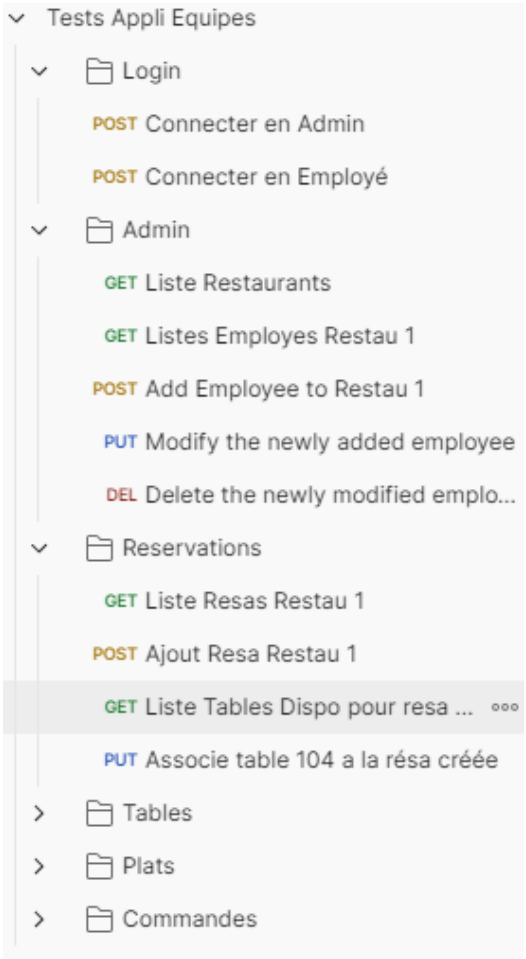


*Figure 37: Collection de requêtes Postman pour nos tests*

Pour la sécurité à proprement parler, toutes nos classes ont été mises dans le package security avec un sous-package dédié à la gestion des tokens JWT. J'ai mis en place la classe SecurityConfig qui nous sert à configurer la manière dont on veut implémenter Spring Security. Je l'ai pour cela annotée de @Configuration et @EnableWebSecurity.

Elle est composée de deux grandes parties. Un filtre, dans lequel on choisit comment traiter nos requêtes http et un choix d'implémentation de certaines classes nécessaires à la connexion.

Pour la mise en place du filtre, j'ai commencé par définir, pour chaque endpoint, quel rôle peut y accéder. On veut en effet, par exemple, restreindre l'accès aux outils d'administration aux utilisateurs ayant le rôle "Admin". J'ai ensuite configuré dans ce même filtre des choses un peu plus fines, comme le fait que notre application gère la connexion de manière stateless ou le fait qu'il faille appliquer le filtre de connexion par token avant celui par login et mot de passe.

Nous avons choisi de ne pas activer la protections CSRF, car cela nécessite de gérer la création et l'envoi d'un token depuis l'application frontend qui atteste à notre backend que les requêtes envoyées viennent légitimement de lui. Cela aurait pris beaucoup de temps pour peu de valeur ajoutée.

```
@Bean  no usages  Purukogi +3
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> {
        auth
                .requestMatchers(HttpMethod.OPTIONS,  ...patterns: "/**").permitAll()
                // login (tout le monde y a accès)
                .requestMatchers( ...patterns: "/login").permitAll()

                // Permettre à l'admin uniquement
                .requestMatchers( ...patterns: "/admin/**").hasAnyAuthority( ...authorities: "ADM")

                // Permettre aux employés uniquement
                .requestMatchers(HttpMethod.GET,  ...patterns: "/tables/**").hasAuthority("EMP")
                .requestMatchers(HttpMethod.GET,  ...patterns: "/reservations/**").hasAuthority("EMP")
                .requestMatchers(HttpMethod.GET,  ...patterns: "/commandes/**").hasAuthority("EMP")
                .requestMatchers(HttpMethod.GET,  ...patterns: "/plats/**").hasAuthority("EMP")

                .requestMatchers(HttpMethod.POST,  ...patterns: "/commandes/**").hasAuthority("EMP")
                .requestMatchers(HttpMethod.POST,  ...patterns: "/commandes").hasAuthority("EMP")
                .requestMatchers(HttpMethod.POST,  ...patterns: "/reservations/**").hasAuthority("EMP")
```

*Figure 38: Une partie du filtre de sécurité de la classe SecurityConfig*

Pour les classes nécessaires à la connexion, il fallait que je choisisse mes implémentations des interfaces AuthenticationManager, AuthenticationProvider, UserDetailsService et PasswordEncoder. Pour ce dernier, j'ai choisi d'utiliser un BCryptPasswordEncoder, car c'est le standard recommandé.

Pour les classes d'authentification, j'ai décidé d'implémenter une version très simple de l'AuthenticationManager qui a une unique délégation à notre AuthenticationProvider. En effet, l'implémentation standard de cet Interface, ProviderManager délègue à une liste d'AuthenticationManager et nous n'avions pas besoin d'une telle machinerie pour notre connexion qui est assez simple.

Pour l'AuthenticationProvider, j'ai choisi d'utiliser un DaoAuthenticationProvider, car il nous permet de préciser un PasswordEncoder et un UserDetailsService par défaut. Mon choix d'implémentation pour ce dernier est juste de passer par la méthode findByLogin du EmployeRepository.

```java
//Instancie un AuthenticationManager dont on précise comment implémenter la méthode authenticate
@Bean 1 usage  Jean Mugniery
public AuthenticationManager authenticationManager() {
    return new AuthenticationManager() {  Jean Mugniery
        @Override  Jean Mugniery
        public Authentication authenticate(Authentication authentication) throws AuthenticationException {
            return  authenticationProvider().authenticate(authentication);
        }
    };
}

//Instancie un AuthenticationProvider pour notre AuthenticationManager
//L'implémentation DaoAuthenticationProvider a besoin d'un UserDetailsService et d'un PasswordEncoder
@Bean 1 usage  Jean Mugniery
AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(userDetailsService());
    authProvider.setPasswordEncoder(passwordEncoder());
    return authProvider;
}

//Instancie un userDetailsService, qui est juste un DAO qui a accès uniquement à findByLogin
//Il sert à récupérer en base l'utilisateur que l'on compare avec les Credentials qui sont dans les objets d'Authentication
//que l'on passe en paramètre de la méthode authenticate que l'on donne à nos classes d'authentification
@Bean 1 usage  Jean Mugniery
UserDetailsService userDetailsService() {
    return String login -> employeRepository.findByLogin(login)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
}
```

*Figure 39: Implémentation de nos classes responsables de l'authentification*

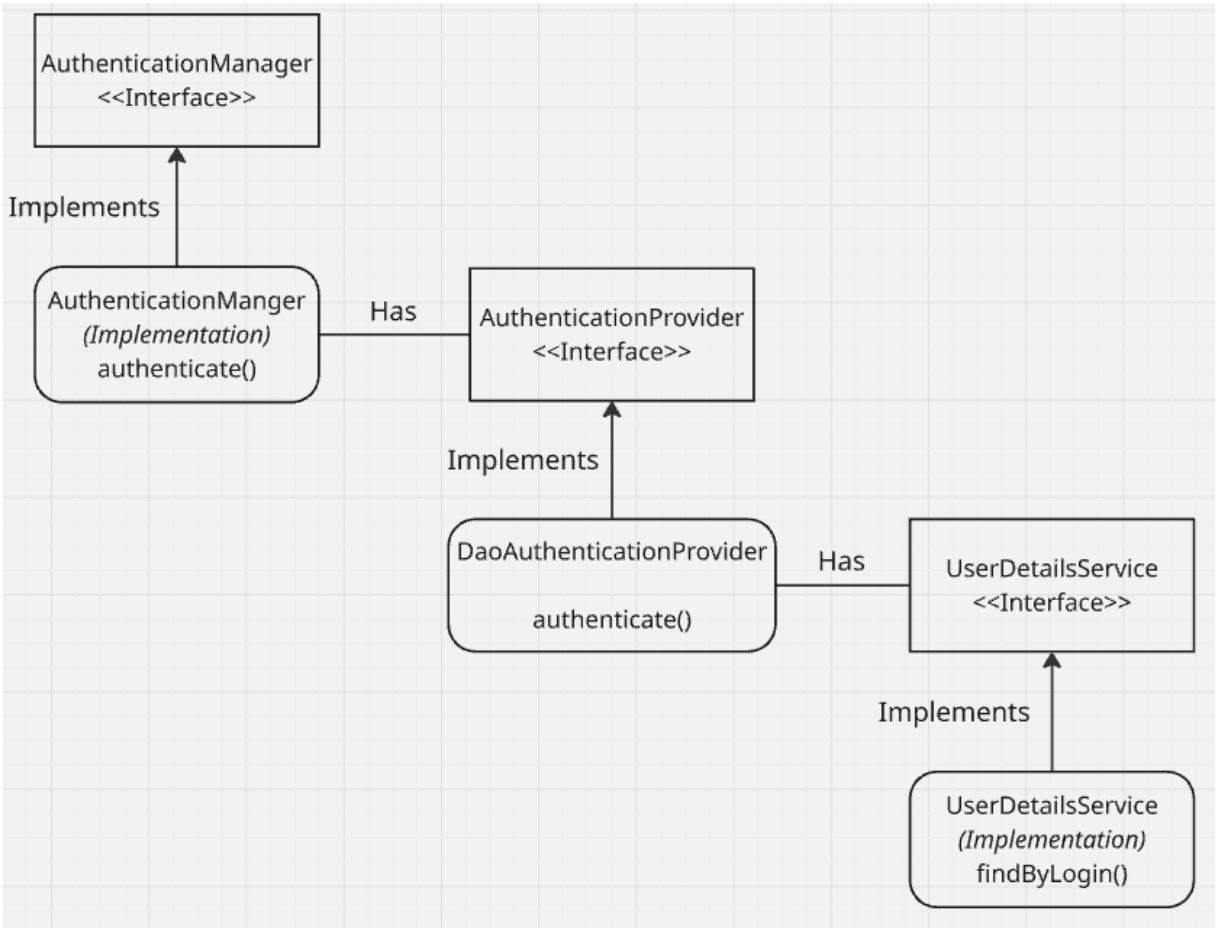


*Figure 40: Schéma des classes d'authentification*

Une fois la sécurité configurée, j'ai travaillé sur les classes permettant aux utilisateurs de se connecter. J'ai commencé par préciser à Spring Security quelle classe correspondait aux users et quels attributs étaient importants pour remplir les objets Authentication que l'on donne à notre AuthenticationManager. Pour cela, j'ai implémenté l'interface UserDetails dans notre classe Employe et j'ai précisé, grâces aux méthodes de l'interface, quels sont les attributs qui correspondent aux username, password et authorities.

```
@Override  Purukogi
public Collection<? extends GrantedAuthority> getAuthorities() {
    return List.of(new SimpleGrantedAuthority(role.getId()));
}

@Override  Purukogi
public String getPassword() { return mdp; }

@Override  Purukogi
public String getUsername() { return login; }
```

*Figure 41: Méthodes explicitant à Spring Security quels attributs correspondent au username, password et authorities de notre UserDetails*

J'ai ensuite mis en place des DTOs pour les entités de connexion : AuthenticationRequest et AuthenticationResponse. Ces classes contiennent pour la première les usernames et password que le frontend envoie et pour la seconde le token JWT contenant les informations de connexion cryptées (username, password, authorities – voir plus loin).

Finalement, j'ai créé les classes AuthenticationController et AuthenticationService. La première est uniquement responsable de l'endpoint POST "/login" qui appelle la méthode login de la seconde.

```
@PostMapping  Purukogi
public ResponseEntity<AuthenticationResponse> login(@RequestBody AuthenticationRequest request) {
    return ResponseEntity.ok(authenticationService.authenticate(request));
}
```

*Figure 42: Méthode de connexion de l'endpoint "/login"*

Celle-ci utilise notre AuthenticationManager pour vérifier si les informations passées dans l'AuthenticationRequest correspondent bien à un utilisateur en base. Si c'est le cas, celui-ci est extrait de l'objet Authentication qui est renvoyé par l'AuthenticationManager, un token JWT est généré, puis inséré dans une AuthenticationResponse qui sera renvoyée, et l'utilisateur connecté est ajouté au contexte de sécurité de l'application.

```java
public AuthenticationResponse authenticate(AuthenticationRequest request) {  1 usage  Purukogi

    //Si l'authenticationManager authentifie un employé en base, on récupère ce dernier
    // dans l'objet Authentication
    Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(request.getLogin(), request.getMdp()));
    Employe employe = (Employe) authentication.getPrincipal();

    //Génère le token
    String jwtToken = jwtService.generateToken(employe);
    AuthenticationResponse authResponse = new AuthenticationResponse();
    authResponse.setToken(jwtToken);
    if("EMP".equals(employe.getRole().getId())){
        authResponse.setIdRestaurant(employe.getRestaurant().getIdRestaurant());
    }

    //Ajoute l'utilisateur connecté au contexte de sécurité
    SecurityContextHolder.getContext().setAuthentication(authentication);

    return authResponse;
}
```

*Figure 43: Méthode authenticate de AuthenticationService*

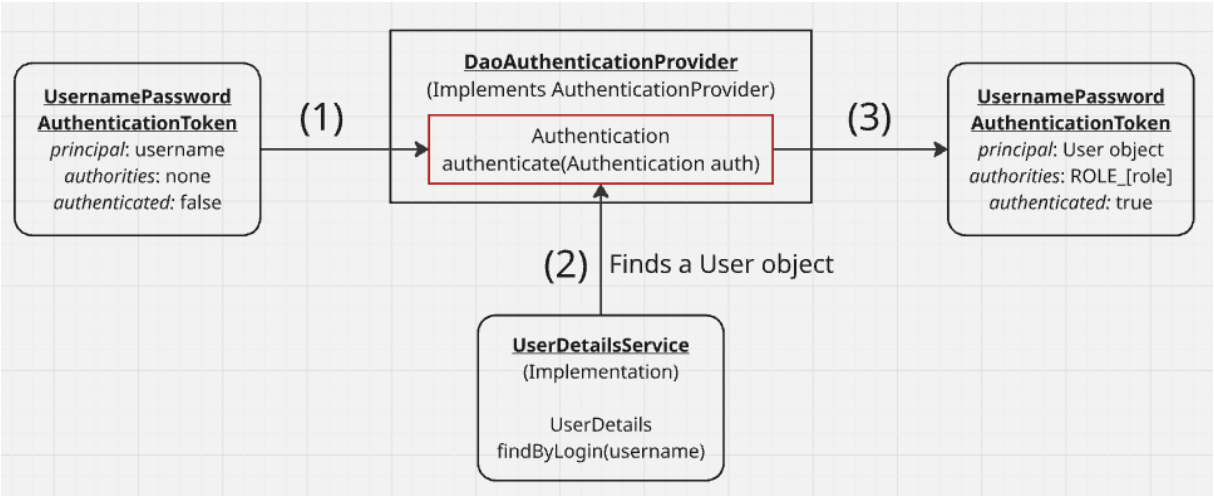


*Figure 44: Schéma de fonctionnement de l'authentification (crédit javainuse.com)*

Pour pouvoir vérifier aisément quels sont les autorisations de l'utilisateur connecté, nous avons aussi mis en place un système de jetons : les tokens JWT. Lorsqu'un utilisateur se connecte, on crée un jeton JWT qui contient ses informations de connexion ainsi que ses rôles. Ces données sont cryptées grâce à une clé secrète qui est cachée dans le fichier application.properties.

Pour gérer ces jetons, j'ai créé 2 classes :

- JwtService, qui regroupe les méthodes nécessaires au traitement des jetons : la génération du jeton, les extractions d'information, la gestion des dates de validité...
- JwtAuthenticationFilter, qui intercepte les requêtes http, et vérifie si celles-ci contiennent bien un jeton JWT. Si c'est le cas, les informations sont extraites, comparées avec celles en base, et si elles correspondent, alors l'utilisateur est connecté et ajouté au contexte de sécurité de l'application.

Avec ces deux classes, nous disposons d'une manière de vérifier les rôles sans avoir à redemander une connexion à chaque requête. Cela conclut la mise en place de Spring Security sur le projet.

```
//nom de méthode héritée de OncePerRequestFilter
public void doFilterInternal(@NonNull HttpServletRequest request, @NonNull HttpServletResponse response,  Purukogi +1 *
                             @NonNull FilterChain filterChain)
        throws ServletException, IOException {

    String authHeader = request.getHeader( name: "Authorization");
    String login = null;
    String jwtToken = null;

    // Check si on a un header qui commence par "Bearer " car le token suit
    if (authHeader != null && authHeader.startsWith("Bearer ")) {
        jwtToken = authHeader.substring( beginIndex: 7); // On récupère juste le token
        login = jwtService.extractLogin(jwtToken);
    }

    // Si on a un utilisateur, mais qu'il n'est pas encore connecté
    if (login != null && SecurityContextHolder.getContext().getAuthentication() == null) {
        UserDetails userDetails = employeRepository.findByLogin(login).orElseThrow();

        // On vérifie si le token est valide avant d'authentifier
        if (jwtService.isTokenValid(jwtToken, userDetails)) {
            UsernamePasswordAuthenticationToken authToken =
                    new UsernamePasswordAuthenticationToken(userDetails,  credentials: null, userDetails.getAuthorities());
            authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
    }
    // Puis, on continue dans la chaine de nos filtres
    filterChain.doFilter(request, response);
}
```

*Figure 45: Filtre de connexion par JWT*

Finalement, je suis aussi intervenu en renfort sur des tickets concernant la gestion des réservations. J'ai commencé par rédiger le endpoint PUT "/tables/{id_table}" qui permet aux équipes soit de passer le statut d'une réservation à "Présent" lorsque des clients ayant réservé arrivent, ou d'accepter des clients arrivant sans réservation.

```java
//Permet d'accepter une nouvelle résa et l'associer à une table
//Deux cas possibles :
//Dans le cas où des gens arrivent sans avoir réservé, le front créé une réservation avec la dateTime.NOW,
//avec un utilisateur nommé "sans_resa" et le statut "Présent" (plus reste du DTO)
//Si la résa est en base, on l'associe juste à la table et on la passe en "Présent"
//Le statut de la table (libre ou occupée) est en fait géré par la requête estLibre de Quentin
@PutMapping("/{id_table}") no usages  Clara LAVIALE +2 *
public ResponseEntity<ReservationDTO> accepterResa(@PathVariable("id_table") Integer idTable,
                                                   @RequestBody ReservationDTO resaAAccepter){

    Reservation resa = reservationMapper.toEntity(resaAAccepter);

    Optional<Restaurant> restaurant = restaurantService.findById(resaAAccepter.getIdRestaurant());
    restaurant.ifPresent(resa::setRestaurant);

    // Associer la table
    resa.setTable(tableService.getById(idTable));

    // Cas sans réservation = nomClient "Sans"
    if ("sans_resa".equalsIgnoreCase(resaAAccepter.getNomClient())) {
        Utilisateur utilisateurParDefaut = utilisateurService.selectByNom("Sans");
        resa.setClient(utilisateurParDefaut);
        resaAAccepter.setId(resa.getIdReservation());
    } else {
        // Cas avec réservation = nom du vrai client
        Utilisateur utilisateur = utilisateurService.selectByNom(resaAAccepter.getNomClient());
        resa.setClient(utilisateur);
    }
    resa.setStatut("Présent");
    reservationService.create(resa);
    return ResponseEntity.ok(resaAAccepter);
}
```

*Figure 46: Passage d'une réservation en « présents »*

J'ai aussi codé les endpoints GET "/reservations/{id_restaurant}/{id_table}" et PUT "/reservations/{id_restaurant}/{id_réservation}". Ces derniers permettent respectivement de renvoyer la liste des tables libres avec le bon nombre de places pour en associer une à une réservation "En attente", puis une fois la table sélectionnée, d'ajouter celle-ci à la réservation et de passer le statut de cette dernière à "Confirmée".

```java
//renvoie la liste des tables libres et avec le bon nombre de places
//pour pouvoir en associer une à une réservation "en attente"
@GetMapping("/{id_restaurant}/{id_reservation}") no usages  Purukogi +1
public ResponseEntity<List<TableRestaurantDTO>> tablesPourResa(@PathVariable("id_restaurant") Integer idRestau,
                                                               @PathVariable("id_reservation") Integer idResa){
    Reservation resaAValider = reservationService.getById(idResa);

    List<TableRestaurantDTO> tables = tableRestaurantService.getTablesLibres(idRestau, resaAValider.getHoraireReservation())
            .stream() Stream<TableRestaurant>
            .filter( TableRestaurant table -> resaAValider.getNbPersonne() <= table.getNbPlaces() + 1)
            .map( TableRestaurant table -> tableRestaurantMapper.toDTO(table)) Stream<TableRestaurantDTO>
            .collect(Collectors.toList());

    return ResponseEntity.ok(tables);
}
```

*Figure 47: Récupération de la liste des tables*

```java
//une fois la table sélectionnée, on l'ajoute à la résa et on passe celle-ci en "Confirmée"
@PutMapping("/{id_restaurant}/{id_reservation}") no usages  Purukogi +1
public ResponseEntity<ReservationDTO> accepterReservation(@PathVariable("id_restaurant") Integer idRestau,
                                                          @PathVariable("id_reservation") Integer idResa,
                                                          @RequestBody TableRestaurantDTO table) {
    Reservation reservation = reservationService.getById(idResa);
    TableRestaurant tableAAjouter = tableRestaurantService.selectByNumeroTableAndIdRestaurant(table.getNumeroTable(), idRestau);

    reservation.setStatut("Confirmée");
    reservation.setTable(tableAAjouter);

    reservationService.create(reservation);

    return ResponseEntity.ok(reservationMapper.toDTO(reservation));
}
```

*Figure 48: Confirmation d'une réservation*

## Conclusion :

Cette application a été assez délicate à réaliser, notamment parce que c'est la première fois que nous travaillions sur une application où le front et le back sont découplés et parce que j'étais responsable de la mise en place de Spring Security sur le projet.

Cependant, parce que nous avons pris du temps dans le sprint planning pour mettre en place les maquettes, définir les endpoints dont nous avons besoin lorsque nous faisions le parcours d'un utilisateur, cela s'est relativement bien passé. Encore une fois, la répartition des tickets par fonctionnalité nous a permis de ne pas nous marcher dessus lors du développement.

Le plus gros challenge que j'ai rencontré pour cette application a été la mise en place de Spring Security sur le projet. C'est une machinerie extrêmement complexe, qui est régulièrement mise à jour. Pour l'implémenter sur le projet, nos formateurs nous ont fourni un squelette du code, mais il a quand même fallu que je me documente pour finir de mettre les choses en place.

Par exemple, il a fallu choisir une implémentation des Interfaces AuthenticationManager, AuthencationProvider, UserDetailsService... De plus, je n'arrivais pas à faire que Spring Security reconnaisse le PasswordEncoder que nous avions choisi, ce qui a conditionné mon choix d'implémentation d'AuthenticationProvider.

Une amélioration qu'on aurait pu mettre en place aurait été d'écrire un contrat d'interface. J'ai découvert cela lors de mon immersion en entreprise. Il s'agit lorsque l'on a accès aux maquettes complètes de définir :

- La liste des webservices dont on a besoin
- Les données dont nos webservices ont besoin, au format JSON ou dans les url
- Les données retournées par nos webservices au format JSON

Cela nous aurait permis de bien savoir de quelles données on a besoin, que ce soit pour le front ou pour le back. Nous avons fait les premiers pas vers cette idée lors de notre sprint

planning, mais nous n'étions pas allés au bout, par manque de visibilité sur la jointure back / front.

Un point négatif a été le manque de temps. J'aurais aimé pouvoir implémenter plus tests et de contrôles sur notre application, mais cela n'a malheureusement pas été possible.

### 4.3.2 Application Frontend

#### Spécifications techniques :

Cette application nécessite de faire appel aux webservices mis en place par la partie backend.
Les maquettes sur lesquelles nous nous sommes basés ont été réalisées avec Figma.
La partie Frontend de cette application a été réalisée avec Angular. Pour développer dans ce framework nous avons utilisé l'IDE Visual Studio Code. Nous avons aussi dû installer Node pour le serveur sur lequel notre application Angular est déployée.
Nous avons décidé d'utiliser Bootstrap pour rendre notre application responsive.

#### Réalisation du projet :


*Figure 49: Architecture frontend application équipes*

Lors de notre sprint planning pour cette application, nous avons commencé par nous mettre d'accord, à partir des maquettes que nous avions réalisé sur Figma, de quels composants Angular nous aurions besoin. Nous avons réparti ces derniers par thèmes (administration, réservations, commandes) et nous nous sommes répartis chacun les tickets correspondant à un thème.

Nous avons réparti nos composants en quatre grands blocs réunissant des fonctionnalités : admin, reservation, commande, global. Dans chacun de ces dossiers nous avons créé les composants que nous avons défini lors du sprint planning, en nous assurant que les éventuels composants enfants se trouvent dans le répertoire de leur parent pour plus de visibilité.

Nous avons aussi créé des dossiers pour nos guards, interceptors, interfaces et services. Nous avons mis en place les objets qu'ils contiennent au fur et à mesure du développement. Les interfaces correspondent aux DTOs que notre backend envoie et les services sont responsables d'appeler les webservices.

## Réalisations personnelles :

Pour cette partie du projet, j'étais responsable des fonctionnalités de connexion et d'administration ainsi que des pages de navigation. Je suis aussi intervenu en renfort sur la mise en place d'un header dynamique. Comme le front de cette application nécessite de faire appel aux webservices du backend, sur lequel nous avons implémenté Spring Security, il a fallu que je commence par mettre en place la connexion, ainsi que le transfert des tokens JWT entre le back et le front.

A noter que nous avons, lors du développement de notre frontend, apporté quelques modifications à notre backend pour accommoder notre travail. Un de ces changements concerne l'objet AuthenticationResponse. En plus du token JWT de l'utilisateur qui se connecte, cet objet stocke maintenant aussi l'id du restaurant dans lequel l'employé travaille (et 0 si on se connecte en admin).

Pour la connexion, j'ai commencé par créer un service authentication. Il contient deux méthodes : connexion() et store_user(). La méthode connexion() prend en paramètre un login et un mot de passe et les envoie au travers un objet JSON au webservice "/login" de notre application backend. La méthode store_user() prend en paramètre une AuthResponse (qui est un Interface Angular qui correspond à un objet AuthenticationResponse de la partie backend et se contente de stocker le jeton JWT que celui-ci contient dans le local storage.

```
export class AuthenticationService {

  constructor(private http : HttpClient) { }

  connexion(login: string, mdp: string) {
    return this.http.post<AuthResponse>("http://localhost:8080/login", {login: login, mdp : mdp});
  }

  store_user(response : AuthResponse): void {
    localStorage.setItem("jwt", response.token);
    localStorage.setItem("id_restaurant", response.idRestaurant.toString());
  }

}
```

*Figure 50: Service d'authentification*

Puis, j'ai mis en place un composant formulaire-connexion. C'est juste un formulaire qui contient un champ login et mot de passe. Lorsque l'utilisateur soumet le formulaire, le composant s'abonne au résultat de la méthode connexion() du service authentication.

Lorsqu'il reçoit une réponse, si le login et mot de passe sont corrects, alors la méthode store_user() est appelée et l'utilisateur est redirigé vers la page de navigation correspondant à son rôle (Admin ou Employé). Dans le cas contraire, un attribut boolean du composant formulaire-connexion incorrect_login passe à true et une balise *ngIf dans le html affiche un message d'erreur "Login ou mot de passe incorrect".


*Figure 51: Formulaire de connexion mobile*

Maintenant que le frontend est capable de récupérer le jeton JWT dont il a besoin pour accéder à nos webservices, il a fallu mettre en place une mécanique lui permettant de faire parvenir ce dernier au backend. Pour cela, j'ai mis en place un Interceptor qui est une classe qui hérite de HttpInterceptorFn et qui intercepte nos requêtes http pour y effectuer un traitement. Dans notre cas, s'il existe un jeton dans le local storage ajoute ce dernier dans l'attribut "Authorization" du header de la requête, sous la forme vue "Bearer [token jwt]".

Cette manière de faire permet d'envoyer à notre backend le token JWT sous une forme qui sera reconnue par le filtre JwtAuthenticationFilter que j'ai mis en place. A noter que pour que les choses fonctionnent comme nous le voulons, il a fallu exclure les requêtes en OPTIONS, ainsi que celle vers l'endpoint "/login" de notre Interceptor.

```
export const authenticationInterceptor: HttpInterceptorFn = (req, next) =>
    const jwt = localStorage.getItem("jwt");

    if (req.method == "OPTIONS") {
        return next.call("handle", req);
    }

    if (jwt) {
        if(req.url != "http://localhost:8080/login"){
            const cloned = req.clone({
                headers: req.headers.set("Authorization",
                    "Bearer " + jwt)
            });
            return next.call("handle", cloned);
        }
    }
    return next.call("handle", req);
};
```

*Figure 52: Interceptor pour faire passer le JWT au backend*

Les pages de navigation admin et employé ont été très simples à mettre en place, ce sont juste des boutons placés en flex sur la page afin de rendre l'affichage responsive. Le travail supplémentaire que j'ai effectué a été de créer des Guards pour m'assurer qu'aucun composant autre que celui de login n'était accessible à un utilisateur non connecté.

J'ai donc mis en place deux Guards auth-admin et auth-employe, que j'ai appliqué aux routes vers les composants admin ou employés qui se trouvent dans le fichier app.routes.ts. L'implémentation de ces deux guards s'effectue de manière similaire, donc je ne développerai que auth-admin. Lorsqu'un utilisateur essaie d'accéder à un composant, s'il existe bien un "id_restaurant" dans le local storage et que celui-ci vaut "0", alors la Guard nous laisse passer. Sinon on est bloqué.

```
export const authAdminGuard: CanActivateFn = (route, state) => {
  const router = inject(Router);

  if (localStorage.getItem("id_restaurant") == null ||  localStorage.getItem("id_restaurant") != "0"){
    router.navigate(['/login']);
    return false;
  }
  return true;
};
```

*Figure 53: Guard pour l'authentification en tant qu'administrateur*

Puis je me suis attaqué aux fonctionnalités de l'administrateur. Lorsque celui-ci accède à la gestion des employés, une liste de ses restaurants s'affiche. Puis, lorsqu'il sélectionne un restaurant, la liste des employés de ce dernier apparait. Il peut alors ajouter un employé grâce à un bouton en bas de l'écran, consulter les détails de l'employé en cliquant sur ce dernier ou supprimer son employé en cliquant sur une icône du composant employé.

La liste des restaurants correspond à un composant Angular. Il contient une liste des restaurants qu'il récupère en s'abonnant au résultat de la méthode getRestaurants() du service restaurant. Cette méthode appelle le webservice "/admin" en GET pour récupérer la liste des restaurants en base. Il affiche une liste de composant restaurant-item, à l'aide d'un *ngFor placé dans la balise du restaurant-item.

Pour afficher nos restaurants, j'ai créé un Interface Restaurant qui correspond au RestaurantDTO de notre backend. Nos restaurant-items récupèrent par hérédité (à l'aide de l'annotation @Input() et l'ajout de [restaurant]="restaurant" dans la balise) un Restaurant qui est stocké en attribut du composant, afin d'en afficher les informations. Lorsque l'on clique sur le composant, une méthode getEmployes() est appelée, qui redirige vers le composant employe-liste en mettant l'id du restaurant sélectionné dans la route.

L'affichage de la liste des employés se fait de manière similaire à celle des restaurants, avec un composant employe-liste et un composant employe-item. Je fais passer un Interface Employe de la liste vers le composant enfant par hérédité afin d'en afficher les informations lorsque l'on clique sur l'employe-item. Ce dernier contient aussi une icône qui, quand on clique dessus, appelle une méthode supprimerEmploye(). On pourrait se contenter d'appeler le webservice "/admin/{id_restaurant}/{id_employe}" en DELETE, mais cela laisserait l'employe-item à l'écran.

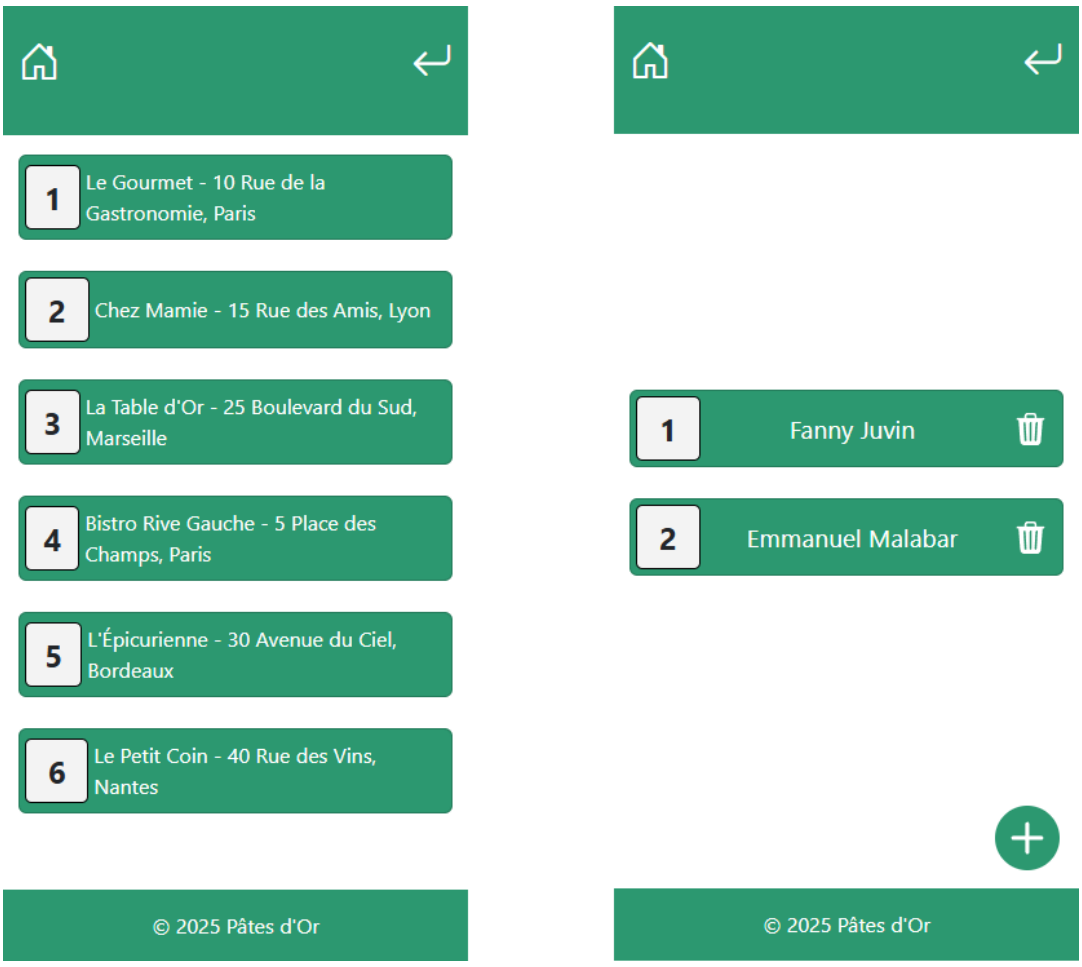


*Figure 54: Affichage des listes de restaurant et d'employés sur mobile*

Pour pouvoir mettre à jour de manière dynamique l'affichage des utilisateurs après suppression, j'ai mis en place un peu de programmation réactive avec une implémentation simple du design pattern observer. J'ai commencé par créer un Interface Observer, qui contient l'unique méthode notify(). Puis, je fais implémenter cet Interface à mon composant employe-list.

J'ai ensuite mis à jour mon Service employe. Je lui ai ajouté une liste d'Observers en attribut et j'ai créé deux méthodes : subscribe() qui prend un Observer en paramètre et qui l'ajoute à la liste des Observers et notify() qui appelle la méthode notify() de chacun des Observers de la liste. Pour que notre liste d'employés observe le Service, nous le faisons s'abonner avec la méthode subscribe() dans la méthode onInit().

Puis, j'ai implémenté la méthode notify() de employe-liste. Cette dernière se contente de refaire un getEmployes() pour refresh la liste des employés qu'elle possède en attribut. Une fois cela fait, alors l'affichage dynamique de la suppression d'employé se fait simplement. Une fois l'employé supprimé de la base de données avec le webservice en DELETE, on notify() le Service qui à son tour notify() la liste.

```
//ajoute un composant pour observer le service
subscribe(observer : Observer){
  this.observers.push(observer);
}

//notifie les observers d'un changement par le service
notify(){
  this.observers.forEach(o => o.notify());
}
```

*Figure 55: Programmation réactive dans le service*

```
supprimerEmploye() {
  this.service.deleteEmploye(this.id_restaurant!, this.employe?.id!).subscribe(
    _ => {
      console.log("Employe supprimé avec succès");
      //On notifie à notre service qu'un changement a été effectué dans la BDD,
      //afin que les observers soient notifiés à leur tour
      //cela leurs permet de mettre à jour leurs données sans avoir à refresh la page
      this.service.notify();
    }
  );
}
```

*Figure 56: Programmation réactive dans le composant*

Notons que comme on a une action à la fois lorsque l'on clique sur employe-item et lorsque l'on clique sur l'icône de suppression située sur l'item, il a fallu que je stoppe la propagation de l'event pour éviter que le formulaire de consultation de l'employé que l'on est en train de supprimer ne s'ouvre.

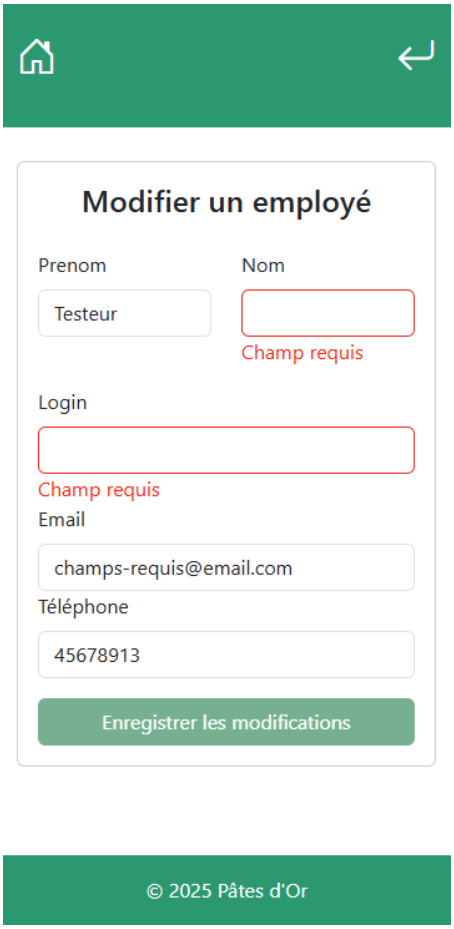


*Figure 57: Formulaire de modification d'employés sur mobile*

Ensuite, pour la gestion des employés, lorsque l'on clique sur employe-item, un formulaire s'ouvre pour que l'administrateur aie accès à ses informations. Il a alors la possibilité de modifier celles-ci grâce au webservice "/admin/{id_restaurant}/{id_employe}" en PUT.

J'ai mis en place un contrôle sur le formulaire, de sorte que les champs obligatoires soient renseignés. S'ils ne le sont pas, un message d'erreur s'affiche. De plus le bouton de soumission du formulaire est disabled tant que le formulaire ne passe pas les vérifications.

```
constructor(private detailService : EmployeDetailsService,
            private service : EmployeService,
            private router : Router){
  this.employe = this.detailService.employe;
  this.id_restaurant = this.detailService.id_restaurant;
  this.formulaire_modification = new FormGroup({
    nom : new FormControl(this.employe?.nom, Validators.required),
    prenom : new FormControl(this.employe?.prenom, Validators.required),
    login : new FormControl(this.employe?.login, Validators.required),
    email : new FormControl(this.employe?.email),
    telephone : new FormControl(this.employe?.telephone)
  });
}
```

*Figure 58: Mise en place de contrôles sur les champs du formulaire*

Sur la page listant les employés d'un restaurant, l'administrateur dispose aussi d'un bouton permettant l'ajout d'un employé. Cela ouvre un formulaire qui, lorsqu'il est soumis, envoie les informations de l'employé en POST au webservice « /admin/{id_restaurant} ». La mise en place des composants et des contrôles a été faite de manière similaire à la modification d'employé.

Enfin, j'ai été appelé à agir en renfort pour mettre en place un header dynamique. En effet, sur ce dernier nous voulions deux features : un bouton de retour arrière et des icônes qui changent en fonction de la page sur laquelle on se trouve.

Pour le premier, j'ai implémenté une méthode goToLastUrl() qui appelle juste this.location.back() pour afficher le composant précédent. Pout les icônes dynamiques, on voulait que le header ne contienne rien lorsque l'on se trouve sur la page de login et que seule l'icône de déconnexion s'affiche lorsque l'on est sur la page de navigation. Pour faire cela, j'ai juste créé deux méthodes : une qui vérifie si notre url est celle de login et une qui vérifie si c'est une page de navigation. Puis j'ai mis en place des *ngIf dans les balises de nos différentes icônes.

## Conclusion :

Encore une fois, un sprint planning solide lors duquel nous avons bien discuté de quels composants nous avions besoin en amont du développement a été d'une grande aide. De plus les maquettes que nous avions réalisées sur Figma nous ont donné un cadre de travail très agréable. Enfin, parce que les fonctionnalités sur lesquelles nous avons travaillé sur la partie front correspondaient à celles que nous avions développé pour la partie back, il était plus simple de mettre nos composants en place.

Je me suis senti beaucoup plus à l'aise avec la partie front de cette application comparée à celle grand public. J'ai commencé à mieux prendre mes aises avec Bootstrap, notamment en commençant à travailler sur un affichage mobile, puis en rendant responsive en basculant vers des écrans plus grands (tablettes, PC). De plus la réalisation à base de composant proposée par Angular m'a beaucoup aidé à mettre mes écrans en place.

Un challenge que nous avons rencontré a été dû au fait que certains de nos endpoints étaient trop compliqués de manière non nécessaire. Par exemple, nos endpoints de gestion d'employés ont la forme "/admin/{id_restaurant}/{id_employe}" et il a donc fallu dans notre front que je fasse parcourir cet id_restaurant au travers mes différents composants.

Une autre difficulté de notre sprint a été due au fait que Spring Security était déjà implémenté sur notre backend. Cela nous a freiné dans le développement de notre front, dans la mesure où nous ne pouvions pas appeler nos webservices tant que je n'avais pas mis en place la connexion et le traitement des token JWT.

Parmi les améliorations potentielles, on aurait pu implémenter la gestion de l'id_restaurant que l'on stocke en local storage pour pouvoir afficher les tables, réservations, ... En effet, avec

notre implémentation, si on modifie manuellement l'id_restaurant en local storage, un utilisateur peut avoir accès aux informations d'un restaurant dans lequel il ne travaille pas.

Une résolution aurait été de stocker cet id_restaurant de manière cryptée dans le token JWT plutôt qu'en clair dans l'AuthenticationResponse qui est renvoyé lors du login. Cependant cela nous aurait demandé de mettre en place une manière de décrypter le token JWT depuis le front pour pouvoir récupérer l'information.

De plus, cela aurait aussi aidé d'un point de vue sécurité puisque nos Guards utilisent l'id_restaurant pour bloquer la navigation. Cependant, pour des contraintes de temps, nous n'avons pas pu implémenter cette manière de faire.

# CONCLUSION

La fin de ce projet fil rouge conclut les quatre mois de formation. Durant cette période, j'ai énormément appris et je suis extrêmement heureux des progrès que j'ai accompli.

Je suis monté en compétences sur plusieurs langages de programmation, en particulier Java, mais aussi pour des technologies frontend avec Angular et Bootstrap. Cela a été une excellente expérience et une agréable surprise car je n'étais pas du tout à l'aise avec le front au début de la formation.

J'ai aussi trouvé très intéressantes les questions de sécurité que nous avons rencontré au cours de nos différentes réalisations, ainsi que l'implémentation de protections. Mettre en place Spring Security sur notre troisième application en particulier a été un challenge, mais cela a été très éducatif.

La (re)découverte du travail en équipe, notamment grâce à notre implémentation des méthodes agiles et lors des sprints que nous avons mis en place a été rafraichissant après plusieurs années de travail plus individuel. Débattre et réfléchir à la mise en place des solutions à mettre en place pour répondre aux besoins du client, et pouvoir réfléchir en équipe à la résolution de nos problèmes collectifs m'a beaucoup plu, et cela a permis de bien cadrer le travail.

Parmi les points négatifs, le manque de temps a été le plus gros facteur de frustration. Ne pas pouvoir aller au bout des projets ou approfondir certaines notions était déplaisant, mais cela est normal pour une formation aussi courte. Je suis tout de même très fier de ce que l'on a accompli durant cette période.

Pour les axes d'améliorations, j'aurais aimé pouvoir mettre plus de tests et de contrôles en place sur la troisième application. J'aurais aussi voulu creuser de manière plus approfondie la sécurité, notamment la gestion des jetons JWT (avec comme discuté, l'inclusion de l'id_restaurant) et la protection csrf. Malheureusement, je n'ai pas pu le faire par manque de temps.

J'aimerais dans le futur aussi retravailler la partie DevOps et CI / CD qui est très riche, mais que nous n'avons que peu abordé lors de la formation.

Je suis de manière générale très content de ce que j'ai appris et ce que nous avons accompli au cours de ces quatre mois, et je suis conforté dans l'idée que le développement est la bonne reconversion pour moi. J'espère pouvoir travailler dans un cadre qui me permettra de continuer à apprendre et à monter en compétences, notamment en sécurité.