Riahi Aboubaker
LMOUDEN OUALID
MEHDI EL GHAZOUANI
Gr : IIR19

Rapport de Projet : Application Web de Commande de Restaurant en Ligne

1. Introduction Générale

Dans le contexte actuel de digitalisation, les restaurants cherchent à proposer des solutions de commande en ligne afin de faciliter l'expérience client et d'optimiser la gestion des commandes. Ce rapport présente une application web de commande en ligne pour un restaurant (notamment une pizzeria), développée avec le framework Django. L'objectif de ce projet est de permettre aux clients de parcourir la carte du restaurant, de personnaliser leurs plats (par exemple choisir la taille d'une pizza ou ajouter des garnitures), d'ajouter des articles à un panier virtuel puis de finaliser leur commande en ligne. Parallèlement, l'application offre aux gérants du restaurant un moyen de gérer le menu (ajout/mise à jour de plats) et de suivre les commandes passées.

Cette application s'inspire d'un cas réel (la pizzeria *Pinocchio's Pizza & Subs*) afin de couvrir un menu complet avec une variété d'options (tailles, garnitures, suppléments, etc.). Elle a été réalisée en utilisant Django côté serveur, une base de données SQLite pour stocker les informations, et des technologies web standard (HTML/CSS, éventuellement Bootstrap) pour l'interface utilisateur. Le présent rapport, rédigé dans un style académique, détaille successivement l'analyse des besoins, l'architecture du système, la conception de la base de données, les fonctionnalités clés de l'application, l'interface utilisateur, les tests et validations effectués, les aspects de déploiement, puis conclut sur les apports du projet et les perspectives envisageables.

2. Analyse des Besoins

Avant la réalisation, une analyse des besoins fonctionnels et non-fonctionnels a été menée afin de définir clairement les attentes du système.

Besoins fonctionnels principaux :
- **Pour les clients (utilisateurs finaux)** : - Consulter la carte du restaurant en ligne, avec la liste des catégories de produits (pizzas, salades, sandwiches, etc.) et les détails de chaque plat (description, options disponibles, prix). - Personnaliser certains articles avant ajout au panier (par exemple, choisir la taille d'une pizza – petite ou grande –, sélectionner des garnitures pour une pizza ou des suppléments pour un sandwich). - Ajouter un ou plusieurs articles au panier virtuel et consulter le contenu du panier à tout moment. - Modifier le panier si nécessaire (ajouter ou retirer des articles, changer une option avant validation). - Créer un compte client et s'authentifier (connexion) pour sauvegarder ses informations et suivre ses commandes. - Valider et soumettre une commande en ligne depuis le panier une fois prêts, à condition d'être authentifié. - Recevoir une confirmation visuelle que la commande a bien été prise en compte (par exemple via l'affichage d'un récapitulatif de commande passée).

- **Pour le gérant/administrateur du site (personnel du restaurant)** :
- Disposer d'une interface d'administration sécurisée pour gérer la carte du restaurant, c'est-à-dire ajouter de nouveaux produits, catégories, options (garnitures, extras) ou mettre à jour les produits existants (nom, description) et leurs prix.

- Pouvoir consulter les commandes passées par les clients. Le personnel doit voir la liste des commandes en attente avec le détail des articles commandés par chaque client.
- Mettre à jour l'état d'une commande (par exemple, marquer une commande comme "complétée" une fois qu'elle a été préparée/livrée).
- Gérer les droits d'accès : seuls les administrateurs doivent avoir accès aux fonctionnalités de gestion du menu et de visualisation des commandes de tous les clients, tandis que chaque client ne voit que ses propres commandes.

Besoins non-fonctionnels :
- **Facilité d'utilisation** : L'interface doit être conviviale, intuitive et responsive (adaptée aux écrans d'ordinateur comme de mobile) pour encourager les clients à l'utiliser. La navigation entre les pages (menu, panier, compte) doit être simple. - **Performance** : Le système doit permettre une navigation fluide dans le menu et un ajout au panier sans latence perceptible. Même si SQLite est utilisé pour le développement, l'architecture doit pouvoir évoluer vers une base de données plus robuste en production sans modifications majeures. - **Sécurité** : La gestion des comptes clients doit respecter des bonnes pratiques (mots de passe stockés de manière sécurisée via Django, pages de connexion/inscription protégées). De plus, les actions sensibles (valider une commande, accéder aux pages d'administration) doivent être réservées aux utilisateurs autorisés (authentification requise, et droits administrateur le cas échéant). - **Fiabilité** : Les données de commande ne doivent pas être perdues. Une fois qu'un client passe une commande, celle-ci doit être enregistrée de façon fiable en base de données et consultable par un administrateur. Le panier d'un utilisateur authentifié devrait persister (les articles restent dans le panier même s'il actualise la page ou se déconnecte et se reconnecte plus tard). - **Évolutivité** : Le système devrait pouvoir supporter l'ajout de nouveaux types de produits ou de nouvelles options sans refonte majeure (par exemple, ajouter une nouvelle catégorie de plat au menu).

En résumé, l'application doit couvrir l'ensemble du processus de commande en ligne d'un restaurant : de la publication du menu jusqu'au suivi de la commande en cuisine, en assurant une expérience utilisateur fluide et une gestion aisée pour l'administrateur.

3. Architecture du Système

L'architecture du système suit le modèle multicouche d'une application web classique en s'appuyant sur le framework Django. On peut la décrire en plusieurs composantes :

- **Architecture client-serveur** : L'application est de type web, ce qui signifie que le client est un navigateur web (interface utilisateur rendue en HTML/CSS/JS) et le serveur est l'application Django qui traite les requêtes HTTP. Le protocole HTTP est utilisé pour la communication entre le client et le serveur. L'utilisateur interagit via des pages web, et chaque action (consultation de page, envoi de formulaire pour ajouter au panier ou passer commande) est traitée par le serveur Django.

- **Framework Django (Modèle-Vue-Gabarit)** : Django utilise un paradigme proche du MVC. On y retrouve :

- Des **modèles (models)** qui représentent les données et la logique métier associée à ces données (structure de la base de données et interactions).
- Des **vues (views)** qui sont des fonctions ou classes Python gérant la logique applicative lors de la réception d'une requête. Elles récupèrent les données nécessaires (depuis les modèles) et déterminent quelle page HTML renvoyer au client.

- Des **gabarits (templates)** qui sont des fichiers HTML dynamiques, combinant de la structure fixe et des données variables injectées par les vues. Ils définissent la présentation de l'interface envoyée à l'utilisateur.

- Le **contrôleur** au sens MVC traditionnel est en partie pris en charge par le routeur URL de Django et les vues elles-mêmes. Les **URLs** sont configurées pour faire correspondre chaque requête à une vue spécifique (via un fichier de configuration des URLs).

- **Séparation en modules Django** : Le projet est organisé en plusieurs modules/applications Django afin de séparer les responsabilités :

- Le module principal de configuration, nommé `pizza`, contient la configuration globale du site (settings Django, définition des URLs globales, configuration WSGI, etc.). C'est en quelque sorte le "cœur" du projet qui lie les différentes applications entre elles.
- L'application `orders` est le module Django dédié à la logique de commande. C'est dans ce module que sont définis les modèles de données relatifs au menu et aux commandes, ainsi que les vues correspondantes (par exemple la vue pour afficher le menu, la vue pour ajouter un item au panier, la vue pour finaliser une commande, etc.). Ce module contient également les templates spécifiques (pages HTML) et éventuellement des fichiers statiques lui étant propres (comme des feuilles de style CSS ou scripts JS liés aux pages de commande).

- Un répertoire `demo` est présent dans les fichiers du projet. Il s'agit possiblement d'un module ou d'un espace de travail utilisé durant le développement initial, mais il n'intervient pas dans les fonctionnalités finales de l'application. L'essentiel de la logique applicative réside dans le module `orders`, tandis que le module `pizza` assure le paramétrage global du projet.

- **Base de données** : Le système s'appuie sur une base de données relationnelle SQLite (fichier `db.sqlite3`). SQLite a été choisi pour sa simplicité dans un environnement de développement, étant donné qu'il ne nécessite pas de serveur de base de données séparé. Django interagit avec SQLite via son ORM (Object-Relational Mapping), ce qui permet de manipuler les données au moyen d'objets Python (instances des modèles Django) sans écrire directement de requêtes SQL. L'architecture est pensée de telle sorte qu'il serait facile de passer à un SGBD plus robuste (PostgreSQL, MySQL) en production, en changeant simplement la configuration Django, sans impacter le code métier.

- **Interface d'administration** : Django fournit par défaut une interface d'administration auto-générée qui permet de réaliser des opérations CRUD (Create, Read, Update, Delete) sur les objets de la base. Cette interface est activée pour ce projet, permettant aux administrateurs du site (le gérant du restaurant) de se connecter à `/admin` et de gérer les modèles définis (catégories, produits, commandes, etc.) via une UI générique. Cette composante s'insère dans l'architecture côté serveur, en offrant un moyen rapide de gérer les données sans devoir coder des vues et templates spécifiques pour chaque opération d'administration.

- **Fichiers statiques et ressources** : L'architecture inclut aussi la gestion des fichiers statiques (feuilles de style CSS, éventuellement fichiers JavaScript et images). Django collecte ces fichiers (par exemple situés dans un répertoire `static/` de l'application) pour les servir au client web. Dans ce projet, des fichiers CSS personnalisés peuvent exister pour styliser l'interface, et possiblement la bibliothèque CSS **Bootstrap** a été utilisée afin de bénéficier de composants visuels réactifs. Les templates HTML font référence à ces ressources statiques pour assurer une présentation agréable.

En synthèse, l'architecture du système est modulaire et repose sur les bonnes pratiques Django : une séparation nette entre la logique métier (modèles et vues Python) et la présentation (templates HTML), une base de données relationnelle accessible via l'ORM, et des outils intégrés (admin, gestion de static) qui accélèrent le développement. Cette architecture assure une base solide et maintenable pour l'application de commande en ligne.

4. Conception de la Base de Données

La conception de la base de données a été guidée par la structure du menu d'une pizzeria et le besoin de suivre les commandes des clients. Le projet utilise la base de données **SQLite** (fichier `db.sqlite3`), et les tables sont créées à partir des **modèles Django** définis dans l'application `orders`. Au total, le modèle de données comporte *huit* entités principales, chacune correspondant à un modèle Django et donc à une table dans la base de données relationnelle. Ces différentes tables et leurs rôles sont décrits ci-dessous :

- **Category** (Catégorie) : Cette table représente les catégories de produits du restaurant. Par exemple, on peut avoir des catégories telles que *Pizza*, *Salade*, *Sandwich*, *Pâtes*, etc. Chaque catégorie sert à regrouper les articles similaires et définit les types de personnalisations applicables. Par exemple, la catégorie *Pizza* permettra d'ajouter des garnitures et de choisir une taille, tandis que la catégorie *Sandwich* permettra des suppléments (comme du fromage supplémentaire). En d'autres termes, les champs de Category peuvent inclure un nom de catégorie et éventuellement des indicateurs sur les options disponibles (une catégorie pizza pourrait avoir un flag "has_toppings" et "has_size", etc.). Les catégories facilitent l'affichage du menu par section et la logique conditionnelle dans l'application (on ne propose la sélection de garnitures que si l'article appartient à une catégorie qui le permet).

- **Size** (Taille) : Cette table liste les tailles possibles pour les produits qui en comportent. Dans le contexte de la pizzeria, deux entrées typiques existent : *Small* (petit format) et *Large* (grand format). D'autres restaurants pourraient en avoir plus, mais ici ce sont principalement les pizzas (ou certaines salades) qui utilisent la notion de taille. Ce modèle très simple sert de référence pour indiquer si un article est petit ou grand, et potentiellement appliquer une tarification différente selon la taille. Chaque enregistrement de Size a un libellé (par exemple "Small" ou "Large"). Les relations: un article (Item) peut être associé à une taille via la tarification (voir Price_List ci-dessous).

- **Topping** (Garniture) : Il s'agit de la liste de toutes les garnitures disponibles pour les pizzas. Par exemple : pepperoni, champignons, oignons, olives, etc. Chaque enregistrement représente un ingrédient qui peut être ajouté en supplément sur une pizza. Cette table est liée aux articles de type pizza : lorsqu'un client compose sa pizza, il peut choisir plusieurs toppings qui figurent dans cette table. Dans la base, Topping peut comporter simplement un identifiant et un nom de garniture. Ces garnitures sont proposées à l'utilisateur lors de la personnalisation d'une pizza dans l'interface.

- **Extra** (Supplément) : De manière analogue aux garnitures, la table Extra liste les suppléments possibles pour d'autres types de plats, notamment les sandwiches (ou subs). Par exemple, *Extra cheese* (fromage en plus) est un supplément courant pour un sandwich. Chaque enregistrement de Extra a un libellé de supplément. Ces extras peuvent être ajoutés aux items applicables (dans Pinocchio's, certains sandwiches pouvaient avoir des extras avec un coût additionnel fixe). La présence de tables séparées Topping et Extra permet de distinguer deux types d'ajouts selon la

catégorie du produit (on n'affichera que la liste des Toppings pour une pizza, et que les Extras pertinents pour un sandwich).

- **Price_List** (Liste de Prix) : Cette table joue un rôle central pour la tarification des produits. Elle définit les prix de base des différents articles du menu, en tenant compte éventuellement de la taille. Chaque enregistrement de Price_List correspond typiquement à une combinaison de catégorie et de taille, avec un prix associé. Par exemple, pour la catégorie *Pizza*, il y aura deux entrées dans Price_List : l'une pour pizza petite (Small) avec son prix de base, et l'autre pour pizza grande (Large) avec son prix de base. De même, pour une catégorie sans notion de taille (par ex *Pâtes*), on pourrait avoir un prix standard (avec peut-être Size nul ou un champ distinct). Dans ce projet, la conception a prévu un champ pour un éventuel supplément de prix si la taille est grande : autrement dit, on peut stocker dans Price_List un prix de base (applicable pour le format Small) et un **supplément** pour le format Large s'il y en a un. Par exemple, une pizza cheese Small coûte 12.00 $, et une Large coûte 17.00 $; cela peut être représenté par un enregistrement Price_List avec base_price=12.00 et extra_price=5.00 pour la taille Large. Cette structure évite de dupliquer les informations et rend le modèle flexible. Chaque Price_List est lié à une catégorie spécifique et éventuellement à une taille (ou contient un champ taille). Ainsi, on peut retrouver le prix applicable à un item donné en cherchant l'entrée correspondant à sa catégorie et taille.

- **Item_List** (Liste d'Articles) : Cette table catalogue tous les articles concrets du menu du restaurant, en associant chacun à sa catégorie et à la tarification appropriée. Par exemple, dans une pizzeria, on peut avoir plusieurs items dans la catégorie *Pizza* : *Cheese Pizza*, *Pizza 1 topping*, *Pizza 2 toppings*, *Special Pizza*, etc., chacun étant un item listé. De même, dans la catégorie *Sandwich*, on énumérera chaque type de sandwich (Steak and Cheese, Chicken Parm, etc.). Chaque enregistrement Item_List contient des informations comme le nom du produit (p. ex. "Cheese Pizza"), une référence à sa catégorie (pour savoir quelles personnalisations s'appliquent), et une référence à l'entrée correspondante dans Price_List qui fournit le prix de base et éventuel supplément. En structurant ainsi, on sépare le nom du produit de son prix, ce qui facilite la mise à jour : par exemple, si tous les pizzas grandes augmentent de prix, on peut ajuster le Price_List une fois sans modifier chaque item. L'Item_List sert aussi à l'affichage du menu sur le site : l'application va parcourir cette table pour montrer tous les produits disponibles aux clients, triés par catégorie.

- **Cart_List** (Panier) : Cette table gère le panier virtuel des utilisateurs. Chaque enregistrement de Cart_List représente un article que **un client spécifique** a ajouté à son panier en attente de commande. Les champs typiques incluent : référence vers l'item (Item_List) choisi, l'utilisateur qui a ajouté l'item (référence vers l'utilisateur/authentification Django), les éventuelles options choisies associées à cet item (par exemple, combien de garnitures et lesquelles, quels extras – cela peut être géré soit par des relations séparées, soit par un champ texte descriptif ou JSON, selon l'implémentation), et possiblement la quantité (bien qu'ici on peut supposer qu'un ajout correspond à une unité; si l'utilisateur veut 2 pizzas identiques, il pourrait soit ajouter deux fois l'item, soit changer une quantité). Le panier est lié à l'utilisateur de sorte que même s'il se déconnecte, les items restent associés à son compte. Lorsque l'utilisateur se reconnecte, l'application peut charger les Cart_List non commandés liés à son compte et ainsi restaurer son panier. Cette persistance est un point important de conception : le panier n'est pas seulement en session volatile, il est stocké en base (ou pourrait l'être dans un cache persistant) pour survivre aux fermetures de session. Dans la logique métier, lorsque l'utilisateur valide sa commande, les enregistrements correspondants dans Cart_List seront marqués comme commandés ou déplacés vers une autre table (voir Order ci-dessous).

- **Order** (Commande) : Ce modèle représente une commande passée et finalisée. Une commande regroupe l'ensemble des articles qu'un utilisateur a décidés d'acheter à un instant T. Les champs de Order incluent une référence au **client** (utilisateur qui a passé la commande), une date/heure de commande, et possiblement un statut (par exemple *en cours*, *terminée*). La relation entre Order et les articles commandés peut s'implémenter de deux façons courantes :

- Soit via une relation 1-n entre Order et Cart_List (en ajoutant un champ foreign key dans Cart_List pointant vers Order lorsqu'un item du panier a été inclus dans une commande). Ainsi, dès qu'une commande est passée, tous les items du panier de l'utilisateur se voient attribuer un numéro de commande, ce qui les fige comme éléments commandés et les distingue des items encore en panier (non commandés). L'administrateur peut alors consulter la table Cart_List filtrée par Order pour voir le détail.
- Soit via une table intermédiaire de détails de commande (parfois appelée OrderItem) qui liste les items commandés avec le numéro de commande, en copiant éventuellement les infos nécessaires (dans ce projet, il est probable qu'on utilise la première méthode pour simplifier).

Dans les deux cas, le modèle Order permet de consulter toutes les commandes déjà passées. L'interface admin ou une page dédiée va s'appuyer sur Order pour lister les commandes. L'administrateur peut modifier le statut (par exemple cocher "completed/terminée" une fois préparée). Marké comme *completed*, la commande n'apparaîtra plus dans la liste des commandes en attente, ce qui simule le flux de traitement réel.

Ensemble, ces modèles assurent une représentation complète du domaine : de la structure du menu (Category, ItemList, PriceList, Topping, Extra) en passant par l'action de l'utilisateur (CartList) jusqu'au résultat final (Order).

Relations et intégrité référentielle : Grâce à Django ORM, chaque relation (clé étrangère) est définie au niveau des modèles. Par exemple, Item_List a une foreign key vers Category et vers Price_List. Cart_List a une foreign key vers Item_List (et peut-être vers Topping/Extra pour stocker les choix, selon l'implémentation) et une vers User (le modèle d'utilisateur de Django, pour associer le panier au compte client). Order a probablement une foreign key vers User également (pour savoir qui a passé la commande). Si un champ dans Cart_List pointe vers Order, c'est une foreign key aussi. Ces relations garantissent qu'on ne peut pas insérer en base des références invalides (par exemple un Cart_List faisant référence à un item qui n'existe pas, etc.). De plus, cela facilite les requêtes complexes : on peut facilement interroger toutes les Cart_List d'un utilisateur donné, ou tous les Item_List d'une catégorie donnée, etc., via l'ORM.

Données initiales et fichiers CSV : Le projet inclut plusieurs *fichiers CSV* qui contiennent les données du menu de la pizzeria, extraites du cas réel (Pinocchio's Pizza & Subs). On trouve notamment : - `products.csv` et `itemlist.csv` : listent les différents articles du menu (noms des pizzas, sandwiches, etc.) avec leur catégorie. - `toppings.csv` : liste toutes les garnitures disponibles pour les pizzas. - `pricelist.csv` : contient les informations de prix (prix de base par catégorie et supplément si applicable). Ces fichiers servent de source de vérité pour peupler la base de données SQLite initiale. Un **script d'importation** nommé `import.py` est fourni dans le projet : c'est un script Python qui lit ces fichiers CSV et crée automatiquement les enregistrements correspondants dans la base de données via Django. Autrement dit, lors du déploiement initial, au lieu de saisir manuellement des dizaines d'entrées via l'admin, le développeur peut exécuter `import.py` pour insérer toutes les catégories, tous les items du menu et leurs prix, ainsi que les toppings et extras, conformément aux fichiers CSV. Ce script garantit que la base de données est remplie avec le menu complet du restaurant dès le départ, ce qui permet de proposer immédiatement l'intégralité des plats aux utilisateurs.

Remarque: Le choix de normaliser les données en plusieurs tables (plutôt qu'une seule table produits) permet une grande flexibilité. Par exemple, si l'on veut ajouter un nouveau type de taille (Medium), ou de nouvelles garnitures, cela ne remet pas en cause la structure globale. De même, la séparation du concept de *Item* et *Price* fait que la mise à jour des prix ou l'ajout de promotions pourrait se faire de façon centralisée. Cette conception de base de données a été réalisée avec le souci de respecter les formes normales tout en collant aux particularités du domaine de la restauration (options de personnalisation et variations de prix).

5. Fonctionnalités Clés de l'application

L'application développée remplit les besoins identifiés en offrant un ensemble de fonctionnalités clés, que l'on peut regrouper ainsi :

- **Inscription et Authentification des utilisateurs** : Un visiteur du site peut créer un compte client en fournissant les informations nécessaires (nom d'utilisateur, mot de passe, adresse email, etc.). L'inscription se fait via un formulaire dédié et les données sont enregistrées de façon sécurisée (Django gère le hachage des mots de passe en base). Une fois inscrit, l'utilisateur peut se connecter à son compte via une page de login. La gestion de session est assurée par Django : après connexion, l'utilisateur est reconnu sur le site, ce qui lui permet d'accéder aux fonctionnalités réservées (comme la constitution du panier et la validation de commande). Il existe également la possibilité de se déconnecter (logout) de manière sécurisée. Cette fonctionnalité s'appuie sur le module d'authentification de Django et garantit que seules les personnes enregistrées peuvent passer commande, afin de pouvoir les identifier pour le suivi.

- **Parcours du menu et ajout d'articles au panier** : L'application permet à l'utilisateur (connecté ou non) de naviguer à travers le menu complet du restaurant. La page principale présente les différentes catégories de la carte (par exemple, une section *Pizzas*, une section *Salades*, etc.), chacune listant les items disponibles avec leurs détails. Pour chaque article, les informations affichées incluent son nom, son prix (ou fourchette de prix selon la taille), et le cas échéant des options de personnalisation. Par exemple, pour une pizza, l'utilisateur verra peut-être un formulaire pour sélectionner Small ou Large, et des cases à cocher ou une liste déroulante pour choisir jusqu'à un certain nombre de garnitures parmi la liste disponible. De même, pour un sandwich, il pourrait y avoir des cases à cocher pour les extras (fromage, etc.). L'utilisateur peut configurer l'article à sa convenance puis cliquer **"Ajouter au panier"**. Cette action déclenche l'enregistrement de l'article (avec les choix effectués) dans son panier. Si l'utilisateur n'est pas encore connecté au moment où il tente d'ajouter un article, l'application le redirigera vers la page de connexion (en l'invitant à se connecter ou s'inscrire) car l'association du panier à un compte est requise. Une fois l'article ajouté, le site peut afficher un message de confirmation ou mettre à jour le compteur d'articles du panier dans la navigation. Le panier est ainsi construit dynamiquement par l'utilisateur, article par article.

- **Gestion du panier (consultation et modification)** : À tout moment, un utilisateur authentifié peut consulter le contenu de son panier. L'application propose une page **Panier** où sont listés tous les articles ajoutés en attente de commande. Pour chaque ligne du panier, on retrouve le nom de l'article choisi, les options sélectionnées (par ex: "Pizza Special – Grande, avec garniture: Pepperoni, Olives"), le prix unitaire (calculé en fonction des options/taille) et éventuellement la quantité. Le sous-total et le total général de la commande en cours peuvent être affichés pour plus de transparence. Sur cette page, l'utilisateur a la possibilité de modifier son panier : il peut supprimer un article qu'il ne souhaite plus (par exemple via un bouton "Retirer" à côté de l'item). S'il souhaite changer une option, il doit en général retirer l'article et le reconstituer, sauf si une

fonctionnalité de modification directe a été implémentée. La page panier est un élément clé pour que l'utilisateur valide sa sélection avant de passer la commande. Une attention particulière a été portée à la persistance du panier : comme mentionné précédemment, le contenu du panier reste stocké en base de données. Ainsi, même si l'utilisateur ferme son navigateur ou se déconnecte puis revient plus tard, il retrouvera les articles précédemment ajoutés. Cette persistance améliore l'expérience utilisateur (ils n'ont pas à tout re-sélectionner) et est rendue possible grâce à la table Cart_List liée au compte utilisateur.

- **Passation d'une commande en ligne** : Lorsque le panier contient au moins un article, le client a la possibilité de **passer commande**. En pratique, sur la page du panier, un bouton du type "Valider la commande" ou "Passer la commande" est proposé. En cliquant dessus, l'utilisateur confirme qu'il souhaite finaliser son achat. Le système va alors créer une nouvelle **commande** dans la base (modèle Order) associée à l'utilisateur. Tous les items présents dans le panier de cet utilisateur sont alors liés à cette commande et considérés comme commandés. Concrètement, selon l'implémentation, cela peut vider le panier (retirer ces items de Cart_List ou les marquer comme appartenant à une commande) pour qu'ils n'apparaissent plus dans le panier en cours. Le client voit alors une confirmation que sa commande a été enregistrée avec succès – souvent sous la forme d'une page récapitulative affichant un numéro de commande et la liste des articles commandés, ainsi que le total payé. À ce stade, du point de vue du client, la commande est terminée; il attend la préparation/livraison. Aucune transaction monétaire en ligne n'est gérée dans cette version du projet (le paiement est supposé se faire hors ligne ou à la livraison, sauf extension). L'important est que la commande soit bien enregistrée et visible côté administrateur.

- **Interface d'administration pour la gestion du menu** : L'application offre aux administrateurs (employés autorisés) la capacité de gérer le menu à travers l'interface admin de Django. Après s'être connectés sur la section administrateur (`/admin`), ils ont accès à des écrans de gestion pour chaque modèle pertinent : Catégories, Articles (Item_List), Garnitures, Suppléments, Tarifs (Price_List) et Commandes. Cela signifie concrètement qu'un administrateur peut ajouter une nouvelle catégorie (ex: créer la catégorie "Desserts"), ajouter un item dans une catégorie (ex: "Tiramisu" dans "Desserts", avec son prix), ou mettre à jour le prix d'un article existant (il peut éditer une entrée Price_List ou Item_List correspondante). Il peut également ajouter de nouvelles garnitures ou extras si le menu évolue. Cette fonctionnalité est cruciale pour que le contenu du site reste à jour sans intervention d'un développeur : le gérant a l'autonomie pour modifier la carte du restaurant en fonction des besoins (changement de prix, ajout de plat du jour, etc.). Les opérations de l'interface admin sont sécurisées (il faut un compte staff avec les permissions adéquates) et logguées par Django. À noter que lors du déploiement initial, le menu complet a été importé via script, mais cette interface permet de le faire évoluer.

- **Suivi des commandes par l'administrateur** : Outre la gestion du menu, l'administrateur doit pouvoir consulter et suivre les commandes passées. Deux moyens sont disponibles : d'une part l'interface d'administration Django propose une vue des objets Order (Commandes) enregistrés. Chaque Order y apparaît avec son identifiant, son utilisateur associé et possiblement un statut. L'admin peut cliquer pour voir le détail d'une commande (la liste des items commandés, via les relations Cart_List ou une liste imbriquée). D'autre part, le projet a prévu une page spécifique (côté site web normal, pas dans /admin) pour que l'administrateur connecté puisse visualiser les commandes. En effet, la documentation du projet mentionne *"Site administrators have access to a page where they can view any orders that have already been placed."* Cela implique qu'une vue et un template dédiés ont été implémentés pour lister l'ensemble des commandes en cours ou passées, sous une forme conviviale. Sur cette page "suivi des commandes", un administrateur voit probablement un tableau de toutes les commandes clients avec pour chacune le nom du client, la date/heure et le contenu. Il est possible que l'interface permette de filtrer ou de

marquer une commande comme *complétée*. Par exemple, un bouton ou une case à cocher “Commande réalisée” pourrait être disponible et, lorsqu’il est actionné, la commande disparaît de la liste des commandes en attente. Cette fonctionnalité de suivi est essentielle pour que le personnel du restaurant puisse utiliser l’application dans son flux de travail quotidien : ils peuvent cocher les commandes au fur et à mesure qu’elles sont traitées en cuisine, assurant une bonne communication entre l’équipe et éventuellement un retour au client (dans une version ultérieure, un email ou notification pourrait être envoyé au client pour l’informer que sa commande est prête, mais dans la version actuelle ce n’est pas implémenté explicitement).

En résumé, ces fonctionnalités clés couvrent l’intégralité du cycle de vie d’une commande en ligne : création du compte, constitution du panier, passage de la commande, et traitement de la commande côté restaurant. Chaque fonctionnalité a été testée et pensée pour répondre aux besoins identifiés tout en offrant une expérience utilisateur satisfaisante.

6. Interface Utilisateur

L’interface utilisateur (IU) de l’application a été conçue de manière à être à la fois ergonomique et esthétiquement sobre, en accord avec une utilisation dans un contexte réel de restaurant en ligne. Elle repose sur les templates Django (fichiers HTML dynamiques) et utilise des éléments HTML/CSS standard, avec possiblement l’intégration de la bibliothèque **Bootstrap** pour accélérer la mise en forme responsive et bénéficier de composants visuels prêts à l’emploi (boutons, formulaires, grilles de mise en page, etc.).

Page d’accueil / Menu : La page principale de l’application est dédiée à la présentation du **menu du restaurant**. En général, cette page sert de vitrine pour l’ensemble des produits disponibles. La mise en page organise le contenu par **catégories** de produits, de sorte qu’un utilisateur voit d’un coup d’œil les sections (Pizzas, Salades, etc.). Chaque catégorie peut être présentée avec un en-tête (par ex. *Pizzas* – et éventuellement une courte description ou image thématique) suivi d’une liste des articles appartenant à cette catégorie. Pour chaque **article** du menu, l’interface affiche son **nom**, son **prix** (ou des prix multiples si l’article existe en plusieurs tailles), et inclut un moyen de le **personnaliser** et de l’**ajouter au panier**. Par exemple, pour une entrée de type Pizza qui existe en Small ou Large, l’interface peut présenter deux boutons radio ou un sélecteur pour la taille, ainsi qu’une liste de cases à cocher pour choisir les garnitures désirées (avec éventuellement une restriction sur le nombre maximal de garnitures gratuites permis, e.g. “jusqu’à 3 garnitures incluses”). Un bouton **“Ajouter au panier”** (souvent avec un symbole de caddie/panier) est présent à côté de l’article : lorsque l’utilisateur a sélectionné ses options et clique ce bouton, le choix est envoyé au serveur pour être mis en panier. Visuellement, l’interface pourrait mettre à jour un indicateur de panier (par exemple, dans le menu de navigation en haut de page, un icône de panier avec le nombre d’articles actuels). La page d’accueil sert ainsi de **catalogue interactif** : elle est centrale dans l’expérience utilisateur, car c’est là que se fait la majeure partie de l’interaction (navigation et choix des plats).

Navigation et éléments communs : Sur toutes les pages du site (templates), une barre de navigation ou un menu supérieur est généralement présent pour orienter l’utilisateur. Celui-ci inclut typiquement : - Un **logo ou titre du site** (par exemple le nom du restaurant) qui peut ramener à la page d’accueil lorsqu’on clique dessus. - Un lien vers la page **Menu** (si on n’est pas déjà dessus, mais en l’occurrence la page d’accueil est le menu). - Un lien ou icône vers le **Panier** courant, visible uniquement si l’utilisateur est connecté (ou visible tout le temps mais menant à la demande de connexion si non connecté). Ce lien affiche souvent un badge avec le nombre d’articles dans le panier. - Des liens d’**authentification** : soit “Se connecter / S’inscrire” quand on est visiteur anonyme, soit “Bonjour [Nom] / Se déconnecter” quand on est connecté. La transition de ces liens se fait dynamiquement en fonction de l’état de la session

utilisateur. - Si l'utilisateur connecté a le statut administrateur, des liens supplémentaires peuvent apparaître, par exemple un lien "Administrer" menant soit à la page de suivi des commandes, soit directement à l'interface Django admin. La présence de ces éléments de navigation cohérents sur toutes les pages facilite le parcours de l'application.

Page Panier : Accessible via le lien ou bouton de navigation "Panier", cette page est cruciale pour récapituler la sélection de l'utilisateur. Le template du panier affiche chaque article sélectionné sous forme de ligne dans un tableau ou une liste, avec ses caractéristiques (nom, options choisies, prix unitaire, quantité, total ligne). Un **total général** est calculé au bas du panier. Pour chaque ligne, l'interface prévoit une action de suppression (par exemple un bouton "Retirer" ou une icône poubelle) permettant de supprimer l'article du panier et de mettre à jour la page instantanément (généralement cela entraîne un rechargement de la page avec mise à jour du contexte du panier via la vue correspondante). Si le panier est vide, la page indique "Votre panier est vide" pour clarifier la situation. Cette page est également le lieu où se trouve le bouton **"Passer la commande"** : ce bouton est mis en évidence lorsque le panier contient au moins un item. Une fois cliqué, il déclenche la finalisation de la commande (côté serveur) et redirige l'utilisateur vers la page de confirmation de commande.

Pages d'authentification (Connexion / Inscription) : Ces pages présentent des **formulaires** permettant respectivement de se connecter et de créer un compte. Le design de ces formulaires est simple et centré, avec des champs pour le nom d'utilisateur et le mot de passe (et confirmation du mot de passe, email, etc. pour l'inscription). Des messages d'erreur s'affichent en cas de problème (par exemple "nom d'utilisateur déjà pris" lors de l'inscription, ou "identifiants invalides" lors d'une tentative de connexion). Une attention particulière est donnée à la validation des formulaires et aux retours visuels, afin d'orienter l'utilisateur en cas de saisie incorrecte. Après connexion, l'utilisateur est souvent redirigé vers la page d'accueil ou la page qu'il consultait initialement.

Page de confirmation de commande : Après la passation d'une commande, l'application peut afficher une page de confirmation. Celle-ci remercie l'utilisateur pour sa commande, affiche un résumé de la commande (identique à ce qui était dans le panier, mais figé comme reçu par le système) et éventuellement fournit un identifiant de commande ou un message de suivi ("Votre commande sera prête dans 20 minutes", etc., si de telles informations sont pertinentes). Cette page indique que le processus s'est déroulé correctement. Elle peut également encourager l'utilisateur à revenir à la page menu ou à se déconnecter, etc.

Interface d'administration (admin Django) : Bien que ce ne soit pas destiné à l'utilisateur final, il s'agit d'une composante de l'interface pour les administrateurs du site. L'admin Django n'est pas customisé dans le code du projet (il s'agit du module fourni par Django). Il présente une page de login séparée pour administrateur, puis une fois connecté, une page d'index listant les modèles administrables (Catégories, Items, Orders, Users, etc.). Chaque section donne accès à des listes filtrables, puis à des formulaires de modification/ajout. Cette interface a un style par défaut (fourni par Django, sobre et efficace). Dans le contexte du rapport, on note que les administrateurs auront principalement deux usages : 1. **Ajouter/éditer des éléments du menu** : via les modèles Category, Item_List, Price_List, Topping, Extra. 2. **Suivre les commandes** : via le modèle Order (et indirectement Cart_List) ou via la page spécifique du site.

Considérations de design et de responsive : Le site étant potentiellement utilisé par des clients depuis des appareils variés, il a été réalisé avec une mise en page adaptable. L'utilisation de Bootstrap (si confirmée) permet que les colonnes de la page menu se réorganisent sur mobile (une seule colonne de liste d'articles au lieu de plusieurs), que les boutons et champs de formulaires soient suffisamment larges et utilisables sur écran tactile, etc. Les couleurs et le thème visuel sont restés sobres (souvent, on utilise le thème par défaut Bootstrap bleu/gris, sauf personnalisation mineure pour coller à l'identité du

restaurant). Aucune image d'illustration n'a été mentionnée explicitement dans le projet, donc l'interface est principalement textuelle, avec peut-être un logo du restaurant en en-tête.

Templates et organisation : Techniquement, les fichiers HTML sont rangés dans un répertoire `templates/` au sein de l'application `orders`. On y trouvera par exemple : `templates/orders/menu.html` (pour la page d'accueil menu), `templates/orders/cart.html` (pour la page panier), `templates/orders/login.html` et `register.html` (pour les pages d'authentification), etc. Un template de base `templates/orders/layout.html` ou `base.html` peut exister, contenant la structure commune (header avec navigation, pied de page) et sur lequel les autres pages se basent en héritant (Django template inheritance). Cette factorisation évite de répéter le code HTML commun. Les fichiers statiques CSS/JS, quant à eux, sont placés dans `orders/static/orders/` par convention, et intégrés dans les pages via la balise `{% static %}` de Django.

En définitive, l'interface utilisateur a été pensée pour être simple d'utilisation pour le client (peu de pages, tout est accessible en quelques clics) et efficace pour l'administrateur. Elle respecte les standards du web moderne et exploite les facilités offertes par Django (templates, static files) pour fournir un rendu cohérent et maintenable.

7. Tests et Validation

Une fois le développement réalisé, une phase de tests et de validation a permis de vérifier que chaque composant de l'application fonctionne conformément aux exigences. Étant donné qu'il s'agit d'une application web, les tests ont été effectués principalement de manière manuelle sur le site en conditions réelles de développement, complétés par quelques vérifications automatiques de la logique métier.

Tests fonctionnels manuels : Plusieurs scénarios d'utilisation ont été simulés pas à pas pour s'assurer du bon comportement du système. Parmi les principaux tests effectués, on peut citer :

- *Test du cycle d'inscription et de connexion* : Création d'un nouvel utilisateur via le formulaire d'inscription avec des données valides, vérification en base de la création de l'utilisateur, puis connexion avec ces identifiants. Ce test garantit que le système d'authentification est opérationnel. Des essais d'erreurs ont également été menés (par exemple, tenter de s'inscrire avec un nom d'utilisateur déjà pris, ou avec un mot de passe trop court, pour voir si les messages d'erreur apparaissent correctement et empêchent la création; essayer de se connecter avec un mauvais mot de passe pour vérifier le message d'échec).

- *Navigation et affichage du menu* : En tant qu'utilisateur (non connecté puis connecté), parcours de la page d'accueil pour vérifier que toutes les catégories et items du menu s'affichent correctement. On s'assure que les prix affichés correspondent bien aux données de la base (par recoupement avec les fichiers CSV ou l'admin). On teste aussi l'affichage des options : par exemple, en cliquant sur un item pizza, voit-on bien la liste de garnitures disponible? Sur un sandwich, les extras sont-ils proposés? Ce test valide que les données importées sont bien prises en compte et que la logique conditionnelle d'affichage (en fonction de la catégorie de l'article) est correcte.

- *Ajout au panier* : Test de l'ajout de différents articles au panier. Par exemple, ajout d'une pizza en sélectionnant 2 garnitures, puis ajout d'une autre pizza avec 3 garnitures, puis d'un sandwich avec un extra. Après chaque ajout, vérification que le nombre d'articles dans le panier se met à jour dans l'interface (si un compteur est affiché) et qu'aucune erreur ne survient. Ensuite,

consultation de la page Panier pour vérifier que les articles y figurent avec les bonnes caractéristiques et le bon calcul de prix (une pizza avec 3 garnitures coûte par exemple le prix de base plus éventuel supplément de taille, plus éventuellement un coût additionnel si trop de garnitures; on vérifie ces règles métier). On teste également la suppression d'un article du panier via l'UI et on s'assure que le panier se met à jour (article retiré, total recalculé). Ce test est crucial pour valider toute la mécanique de Cart_List et les vues associées.

- *Persistance du panier* : Un test spécifique a consisté à vérifier que le panier persiste bien d'une session à l'autre. Concrètement, un utilisateur connecté ajoute des items puis se déconnecte ou ferme le navigateur, puis se reconnecte plus tard. On s'attend à retrouver les mêmes items dans son panier (sauf s'il a passé commande). Ce comportement a été observé et validé, confirmant que le panier est bien stocké en base (ou en session persistante liée à l'utilisateur) et non simplement en mémoire volatile de session.

- *Passation d'une commande* : Simulation de la procédure de commande complète. Avec un panier contenant plusieurs articles, l'utilisateur clique sur "Passer la commande". On a vérifié que le système ne permet de le faire que si l'utilisateur est connecté (si on force l'URL de commande sans être connecté, l'application doit rediriger vers la connexion — ce qui a été testé également pour la robustesse). Une fois la commande passée, on vérifie que l'application affiche bien une confirmation. On a également inspecté la base de données (via l'interface admin ou la console Django) pour s'assurer qu'une nouvelle entrée Order a été créée, liée au bon utilisateur, et que les items du panier ont été rattachés à cette commande (et éventuellement retirés du panier actif). On confirme aussi que le panier de l'utilisateur est désormais vide (afin d'être prêt pour de futures commandes). Ce test valide l'intégrité du processus d'achat.

- *Accès administrateur et gestion du menu* : En se connectant avec un compte administrateur (créé via Django admin en coulisse), on a testé l'accès à l'interface d'administration du site. Vérification que l'URL `/admin` est bien protégée (demande un login admin). Une fois connecté, test de l'ajout d'un nouvel item via l'admin : par exemple ajouter une nouvelle garniture, ou un nouveau plat, et enregistrer. Puis, côté site public, vérification que ce nouvel élément apparaît bien (ex: la nouvelle garniture est proposée dans la liste lors de la personnalisation d'une pizza, ou le nouveau plat apparaît dans la bonne catégorie du menu). Test de modification de prix via l'admin (changer le prix d'une entrée Price_List) et contrôle que l'affichage du prix sur le site se met à jour. Ces tests assurent que l'admin peut effectivement gérer le contenu dynamique du site et que les vues publiques réagissent aux changements de données.

- *Suivi des commandes côté admin* : Après avoir passé une commande test, on vérifie que l'administrateur peut la consulter. Dans l'interface admin Django, la commande apparaît bien dans la liste des Orders. On teste également la page dédiée (si implémentée) listant les commandes : en se connectant avec l'admin sur le site normal, on navigue vers la page des commandes (une URL du type `/orders` ou accessible via un lien "Commandes" réservé aux admins). On s'assure que la commande test passée figure dans la liste avec les bons détails. Ensuite, on utilise la fonctionnalité de *marquage comme complétée* (par exemple en cochant une case ou via l'admin Django en modifiant le champ statut). On vérifie que cette action est bien prise en compte : la commande n'apparaît plus dans les commandes en attente (ou son statut visuel change). Si la page admin dédiée actualise en temps réel l'état, c'est validé. Ce test garantit que le restaurant pourra suivre et gérer les commandes reçues.

- *Tests de non-régression et d'erreurs potentielles* : On a également essayé de tester des cas limites ou des comportements inattendus, par exemple :

- Tenter d'accéder à l'URL d'ajout au panier directement sans passer par le formulaire (vérifier que la vue gère l'absence de données requises proprement).
- Soumettre le formulaire d'ajout avec un nombre de garnitures dépassant la limite (si la limite n'est pas aussi contrôlée côté client, le serveur doit idéalement le contrôler).
- Créer deux comptes avec le même email (normalement autorisé par Django par défaut, sauf contrainte ajoutée).
- Charger le site sur un navigateur mobile pour vérifier le comportement responsive (les éléments se réorganisent correctement, pas de barre de défilement horizontale, etc.).
- Vérifier que seules les personnes autorisées accèdent à certaines vues : par exemple, si un utilisateur non admin tente d'accéder à la page de suivi des commandes (en entrant l'URL), il doit être refusé (redirection ou message d'erreur).
- S'assurer que le bouton "Passer commande" n'apparaît pas ou n'est pas actif si le panier est vide.
- Tester le comportement du site avec la base de données initiale vs. après modifications (juste pour s'assurer que l'import initial n'a pas créé de duplicats, etc.).

Résultats des tests : Globalement, les tests effectués ont confirmé que l'application remplit correctement les fonctionnalités attendues. Les utilisateurs peuvent naviguer, s'inscrire, composer leur commande et la finaliser sans rencontrer de bug bloquant. Le panier persistant fonctionne comme prévu, améliorant l'UX. Côté administrateur, la gestion du contenu et le suivi des commandes sont opérationnels. Les quelques ajustements réalisés durant la phase de test ont concerné l'interface (par exemple, amélioration de messages d'erreur, ajustement mineur de style CSS) et de petites corrections logiques (par exemple, s'assurer qu'un utilisateur non connecté ne puisse pas accéder à la page panier sans être redirigé). Aucun problème majeur d'intégrité des données n'a été relevé : la structure en base de données supporte bien les cas testés, et l'ORM Django prévient de nombreuses erreurs en gérant les transactions.

Il n'y a pas eu de suite de tests automatisés (tests unitaires ou d'intégration via un framework comme Selenium ou Django TestCase) fournie dans le cadre de ce projet, mais l'ensemble des tests manuels couvrant les scénarios principaux a permis de valider le système. Pour un déploiement en production, il serait recommandé d'écrire des tests automatisés supplémentaires, mais pour le contexte académique de ce projet, la validation manuelle a suffi à démontrer le fonctionnement conforme aux spécifications.

8. Déploiement

Le déploiement de l'application a été envisagé principalement dans un environnement de développement local, mais les étapes suivies seraient similaires pour une mise en production sur un serveur. L'application étant réalisée avec Django, elle bénéficie d'un serveur web de développement intégré et d'une gestion aisée des dépendances via pip.

Prérequis techniques :
- Disposer d'une installation de **Python** (version 3.x) sur la machine hôte. - Installer le framework **Django** ainsi que les autres bibliothèques nécessaires. Le projet inclut un fichier `requirements.txt` listant les dépendances (principalement Django dans une version spécifique, possiblement d'autres packages s'il y en avait, mais ici ça semble minimal).
- Avoir accès aux fichiers du projet (l'ensemble du code source, incluant `manage.py`, les répertoires `pizza` et `orders`, etc., ainsi que la base de données SQLite ou les CSV pour la recréer).

Installation des dépendances : On commence par créer un environnement virtuel Python (optionnel mais recommandé) puis on exécute la commande d'installation :

```
pip install -r requirements.txt
```

Celle-ci va installer Django et toute autre librairie listée. Dans ce projet, cela installe Django, permettant ainsi d'utiliser les commandes `manage.py` et de faire tourner le serveur.

Mise en place de la base de données : Deux approches sont possibles : - Utiliser la base SQLite fournie (`db.sqlite3` déjà remplie suite à l'import des CSV). Dans ce cas, il suffit de s'assurer que ce fichier est présent à la racine du projet Django. Aucune migration n'est nécessaire puisque la base contient déjà les tables. On peut directement passer à l'étape de création d'un superutilisateur. - Ou bien recréer la base de zéro : par exemple si on souhaite importer les données dans un autre SGBD ou simplement repartir frais. Il faut alors exécuter les migrations Django : `python manage.py migrate` qui va créer les tables vides selon les modèles. Ensuite, exécuter le script d'importation : `python import.py` (ou ouvrir un shell Django et lancer manuellement le chargement des CSV via ce script) pour remplir les tables avec les données initiales du menu. Cette procédure est documentée dans le projet pour reproduire l'état initial.

Dans tous les cas, il faut **créer un compte administrateur** (superuser) pour accéder à l'interface d'administration. Cela se fait avec la commande :

```
python manage.py createsuperuser
```

Django invite alors à entrer un nom d'utilisateur, un mot de passe, etc. pour le superuser. Ce compte servira au gérant du restaurant pour se connecter à /admin.

Lancement du serveur de développement : Une fois l'environnement configuré et la base de données prête, on lance l'application en utilisant la commande Django habituelle :

```
python manage.py runserver
```

Par défaut, le serveur se lance en local sur l'adresse `http://127.0.0.1:8000/`. Le terminal affiche un message confirmant le démarrage ("Starting development server at 127.0.0.1:8000"). On peut alors ouvrir un navigateur web et accéder à l'URL affichée. Le site web de commande en ligne doit alors apparaître, prêt à être utilisé. Durant le développement, ce serveur recharge automatiquement l'application en cas de modification du code, ce qui est pratique pour les ajustements.

Accès à l'interface d'administration : Avec le serveur en marche, on peut naviguer vers `http://127.0.0.1:8000/admin/`. Le navigateur affichera la page de login admin. On se connecte avec le compte superuser créé plus tôt, et on accède ainsi au tableau de bord d'administration de Django. Il est conseillé de tester cela localement pour s'assurer que les modèles sont bien enregistrés dans l'admin (normalement, l'application `orders` a un fichier `admin.py` qui enregistre les modèles Category, ItemList, Order, etc., pour qu'ils soient visibles et modifiables dans l'interface admin).

Déploiement en production : Bien que le projet soit principalement présenté dans un contexte local/universitaire, il est tout à fait envisageable de le déployer sur un serveur en ligne (par exemple Heroku, PythonAnywhere, ou un serveur VPS). Pour ce faire, quelques étapes supplémentaires seraient nécessaires : - Configurer le projet pour un environnement de production : définir `DEBUG = False` dans les settings, configurer `ALLOWED_HOSTS` avec le nom de domaine ou l'IP du serveur, paramétrer

un service de base de données de production (PostgreSQL est recommandé avec Django) et migrer les données (possibilité d'importer les CSV dans la nouvelle base également). - Utiliser un serveur d'application WSGI comme Gunicorn couplé à un serveur web (nginx/Apache) pour servir Django, et s'assurer de la configuration des fichiers statiques (collectstatic pour rassembler les fichiers CSS/JS dans un répertoire servi par nginx). - Mettre en place éventuellement un module de courriels si on voulait envoyer des emails de confirmation, et un système de paiement en ligne si l'on voulait étendre les fonctionnalités. - Pour la simplicité du déploiement, Heroku par exemple permet de pousser le code et utilise le `Procfile`, etc. Dans le projet actuel, un fichier `Procfile` ou équivalent n'a pas été mentionné, donc sans doute l'accent est mis sur le déploiement local.

Documentation et support : Le fichier README du projet (non inclus tel quel dans ce rapport pour ne pas mentionner l'origine) fournit des instructions concises sur comment installer et lancer le projet, que nous avons synthétisées ci-dessus. En cas de difficultés, la documentation Django est une ressource précieuse, notamment pour la configuration de la base de données ou du serveur en production.

En conclusion de la partie déploiement, l'application est relativement simple à mettre en œuvre localement. Avec Python, Django et SQLite, il n'y a pas de dépendance lourde. Cette portabilité permet à un développeur ou un examinateur de facilement exécuter le projet pour évaluation. Sur un serveur de production, quelques configurations additionnelles seraient nécessaires, mais l'architecture Django choisie les supporte sans problème.

9. Conclusion

Ce projet d'application web de commande en ligne pour restaurant a abouti à la création d'un système complet et fonctionnel, couvrant depuis la gestion du menu jusqu'à la finalisation des commandes clients. À travers une analyse méthodique des besoins, une architecture solide basée sur Django et une conception soignée de la base de données, l'application répond aux objectifs initiaux : offrir aux clients une plateforme simple pour commander leurs plats préférés en ligne, tout en fournissant aux gérants du restaurant des outils pour administrer le menu et traiter les commandes efficacement.

Le développement avec Django a permis de tirer parti de fonctionnalités intégrées telles que l'authentification, l'interface d'administration et l'ORM, accélérant ainsi la réalisation du projet et assurant une certaine fiabilité (beaucoup de composants robustes étant fournis par le framework). La structure modulaire (`pizza` pour la configuration, `orders` pour l'application métier) offre une bonne maintenabilité et la possibilité d'étendre le projet. Par exemple, on pourrait aisément ajouter une nouvelle application Django si le restaurant voulait inclure un système de réservation de tables ou de livraison avec suivi, grâce à cette base bien organisée.

Lors des tests, l'application s'est montrée conforme aux attentes : les clients peuvent personnaliser leurs pizzas ou sandwiches, le panier fonctionne de manière persistante, et le passage de commande génère correctement les enregistrements nécessaires pour le suivi. Les administrateurs peuvent mettre à jour la carte et voir les commandes en cours, ce qui est essentiel pour un usage réel.

En termes d'expérience utilisateur, l'interface est simple à naviguer et cohérente. L'utilisation potentielle de Bootstrap a contribué à la rendre responsive, ce qui est important si l'on envisage que de nombreux clients commandent depuis leur smartphone. Le site reste dépourvu d'éléments superflus, focalisant l'attention sur le menu et le processus de commande, ce qui est adéquat pour ce type d'application.

Ce projet, bien qu'accompli dans un cadre académique, pourrait servir de base à une application déployable en conditions réelles avec quelques ajustements. Parmi les améliorations et perspectives

possibles, on peut citer : - **Intégration d'un système de paiement en ligne** : Actuellement, la commande s'arrête à la validation, supposant un paiement hors ligne. Ajouter un module de paiement sécurisé (par exemple via Stripe ou PayPal) permettrait de boucler le processus et de rendre l'application directement utilisable commercialement. - **Notifications aux clients** : Une fois qu'une commande est marquée comme complétée par l'administrateur, l'envoi d'un email ou d'une notification au client (s'il est en attente d'une livraison ou d'un retrait) améliorerait le service. - **Améliorations de l'interface** : Par exemple, afficher des images appétissantes pour chaque plat pourrait rendre l'application plus attractive. De même, une meilleure indication des limites (ex: empêcher côté client de cocher plus de 3 toppings, ou afficher le supplément de prix en temps réel lors de la sélection) rendrait l'usage plus clair. - **Support multi-langue** : Étant en Django, le site pourrait être internationalisé pour toucher une clientèle plus large (anglais/français par exemple). - **Scalabilité** : Si le restaurant grandit ou si le service doit accueillir de très nombreux utilisateurs simultanés, migrer la base de données vers PostgreSQL et déployer l'application sur une plateforme scalable serait à prévoir. La structure actuelle le permet sans refonte majeure. - **Logs et monitoring** : En production, on ajouterait des outils de suivi (logs des commandes, analytics sur les plats les plus commandés, etc.) ce qui pourrait fournir de la valeur au restaurateur.

Pour conclure, le projet **OnlineRestaurantWebApp** (Application web de restaurant en ligne) a atteint ses objectifs pédagogiques et techniques. Il constitue un prototype convaincant de ce à quoi pourrait ressembler le système de commande en ligne d'un restaurant local, aligné sur un exemple réel (Pinocchio's Pizza & Subs) pour coller à un cas d'usage authentique. Le rapport a détaillé chaque aspect du développement – des besoins initiaux à la mise en œuvre – dans un style académique, démontrant la compréhension des enjeux et des solutions apportées. Ce travail pourrait servir de fondation à d'autres développements ou être présenté comme un cas d'étude réussi de réalisation d'application web full-stack avec Django dans le domaine de la restauration. En somme, l'expérience a permis de mettre en pratique des compétences variées (analyse, conception BD, développement backend/frontend, tests, déploiement) et illustre comment celles-ci convergent pour livrer une application web opérationnelle répondant à des besoins concrets.
