

PARCOURS OPENCLASSROOMS

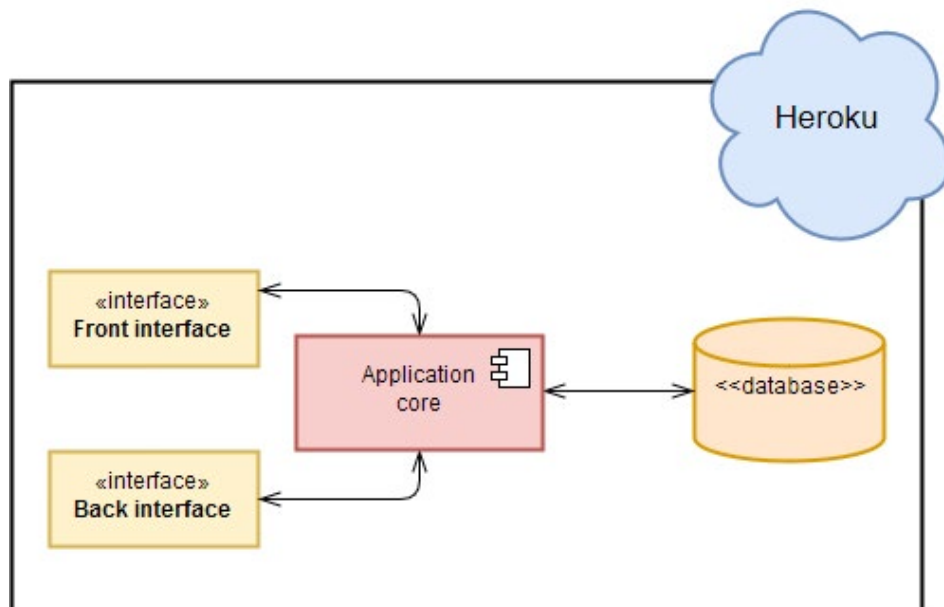
DÉVELOPPEUR D'APPLICATIONS

PYTHON

QUENTIN LATHIÈRE

Projet 6 : Concevez la solution technique d'un système de gestion de pizzeria

https://github.com/Synkied/OC_Projet-6





Le présent document résume la réalisation du 6^e projet (Concevez la solution technique d'un système de gestion de pizzeria) dans le cadre de mon parcours Développeur d'application - Python.

Versions

Auteur	Date	Description	Version
Quentin	10/01/2018	Création du document	1
Quentin	12/01/2018	Ajout des cas d'utilisation	2
Quentin	16/01/2018	Ajout description BDD	3
Quentin	18/01/2018	Ajout diagramme de déploiement	4
Quentin	20/01/2018	Modification description BDD	4.1
Quentin	21/01/2018	Ajout modules à utiliser	5
Quentin	24/01/2018	Réorganisation du document	6

Table des matières

I. Objet du document	4
Objet	4
II. Besoin du client	4
Contexte	4
Enjeux et objectifs	4
Prospection	4
III. Architecture technique	5
1. Application web	5
IV. Architecture de l'application	6
V. Architecture de base de données	7
VI. Architecture de déploiement	13
a. Interfaces	13
b. Terminaux recommandés	13
c. Application	14
d. Serveur de l'application	14
e. Serveur de base de données	14
VII. Acteurs externes	15
1. Paiements:  stripe	15
2. Notifications:  twilio	16
3. Maps: Google Maps	18



I. Objet du document

Objet

Le présent document constitue le dossier de conception technique de l'application OC Pizzeria. Ce document a pour but d'analyser la demande du client OC Pizza afin de :

- modéliser les objets du domaine fonctionnel ;
- identifier les différents éléments composant le système à mettre en place et leurs interactions ;
- décrire le déploiement des différents composants envisagés ;
- élaborer le schéma de la base de données.

II. Besoin du client

Contexte

« OC Pizza » est un jeune groupe de pizzeria en plein essor et spécialisé dans les pizzas livrées ou à emporter. Il compte déjà 5 points de vente et prévoit d'en ouvrir au moins 3 de plus d'ici la fin de l'année.

Enjeux et objectifs

Un des responsables du groupe a pris contact avec nous afin de mettre en place un système informatique, déployé dans toutes ses pizzerias et qui lui permettrait notamment :

- d'être plus efficace dans la gestion des commandes, de leur réception à leur livraison en passant par leur préparation ;
- de suivre en temps réel les commandes passées et en préparation ;
- de suivre en temps réel le stock d'ingrédients restants pour savoir quelles pizzas sont encore réalisables ;
- de proposer un site Internet pour que les clients puissent :
 - passer leurs commandes, en plus de la prise de commande par téléphone ou sur place,
 - payer en ligne leur commande s'ils le souhaitent – sinon, ils paieront directement à la livraison ;
- modifier ou annuler leur commande tant que celle-ci n'a pas été préparée ;
- de proposer un aide mémoire aux pizzaiolos indiquant la recette de chaque pizza ;
- d'informer ou notifier les clients sur l'état de leur commande.

Prospection

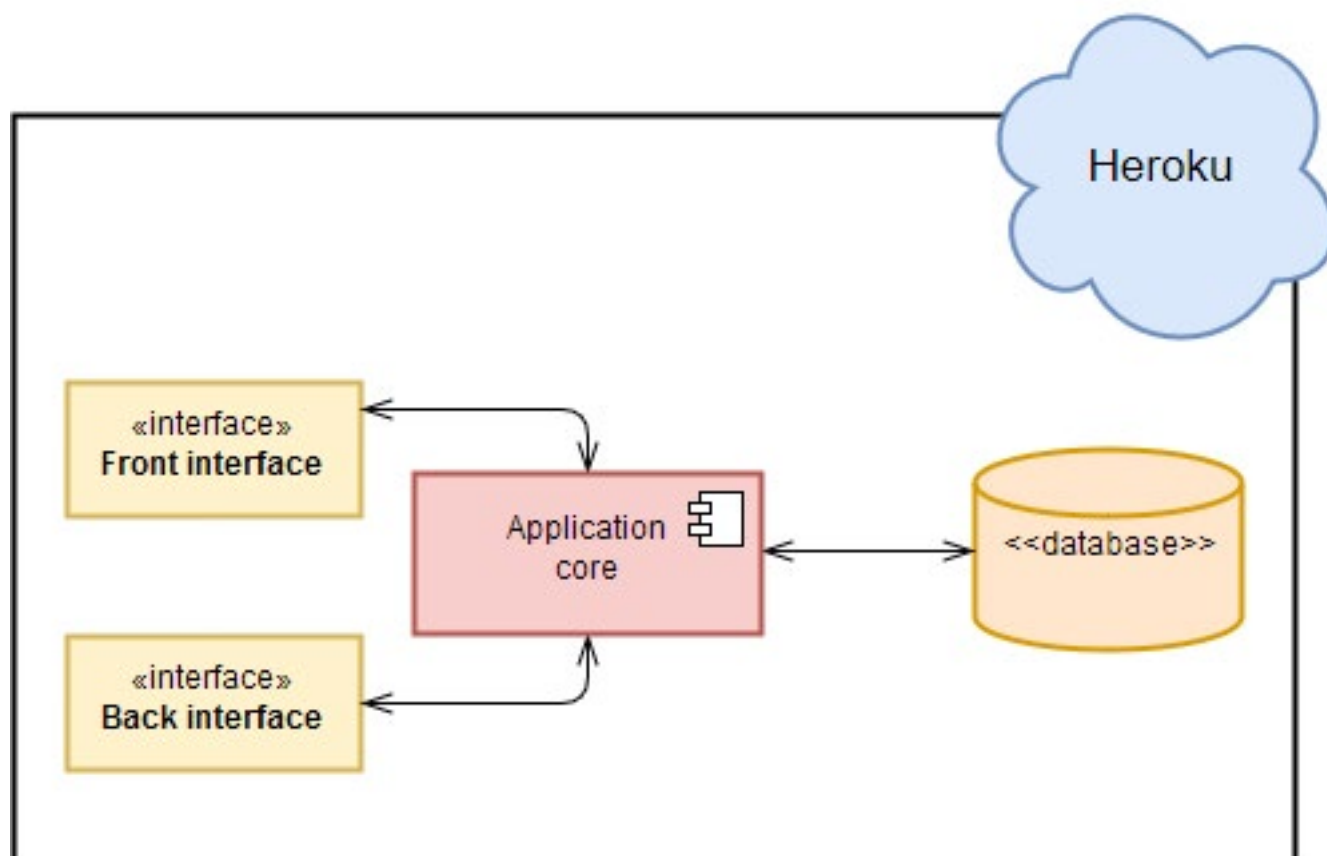
Le client a déjà fait une petite prospection et les logiciels existants qu'il a pu trouver ne lui conviennent pas.



III. Architecture technique

Le logiciel :

- sera composée de 2 interfaces différentes, une interface *front* (client) et une interface *back* (employés) ;
- se basera sur un socle technique et une base de données commune ;
- sera hébergé à l'aide d'une solution cloud flexible de type Heroku.



Avantages/inconvénients

L'avantage d'une telle solution est de séparer les interfaces du cœur de l'application et ainsi en faire une application modulaire et extensible.

De plus, cela en fait une solution bien plus simple à maintenir dans le temps avec les évolutions techniques futures.

Il y a aussi un avantage non négligeable à déployer l'application sur un cloud, car cela ôte la contrainte budgétaire et temporaire d'installation et gestion de serveurs.

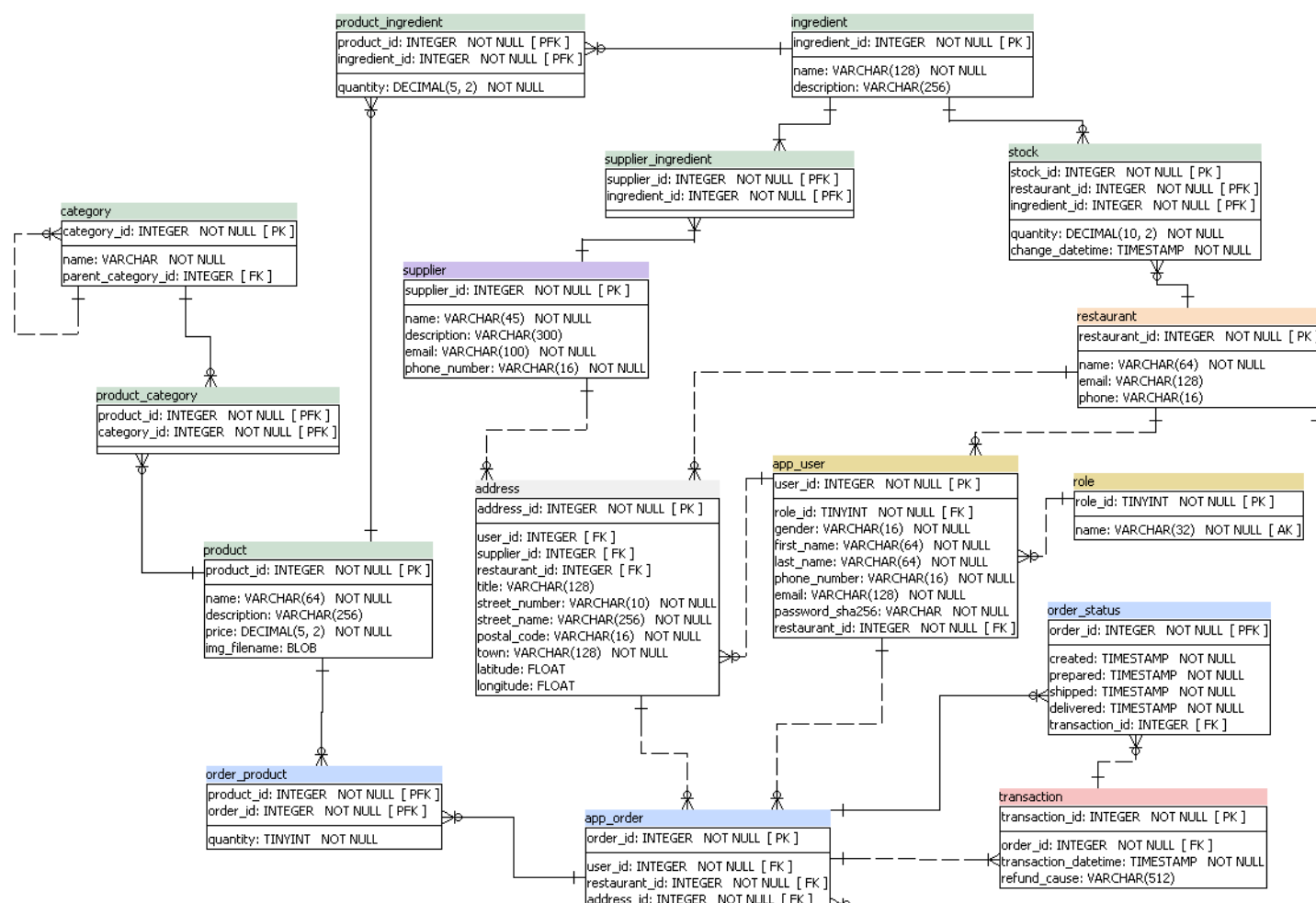
L'inconvénient étant de dépendre alors d'un prestataire tiers et de son bon vouloir.

Dans ce document, le domaine fonctionnel (modèle physique de données) sera tout d'abord présenté, puis seront présentés les aspects d'architecture logicielle et de déploiement.





V. Architecture de base de données



La base de données est composée des tables présentées sur l'image ci-dessus.

Sur les pages suivantes, vous retrouverez un aperçu de chaque table et une description des points importants ainsi qu'une explication des clés étrangères.

Pour une meilleure résolution d'affichage de ce diagramme, un fichier PNG est fourni (OC-Pizza PDM diagram.png)



Table «app_user»

app_user
user_id: INTEGER NOT NULL [PK]
role_id: TINYINT NOT NULL [FK]
gender: VARCHAR(16) NOT NULL
first_name: VARCHAR(64) NOT NULL
last_name: VARCHAR(64) NOT NULL
phone_number: VARCHAR(16) NOT NULL
email: VARCHAR(128) NOT NULL
password_sha256: VARCHAR NOT NULL
restaurant_id: INTEGER NOT NULL [FK]

User:

Permet de stocker tous les utilisateurs de l'application.

role : l'utilisateur a un rôle qui permet de définir ses accès au sein de l'application.

password_sha256 : un hash du mot de passe est stocké en base, pas le mot de passe directement.

FK: restaurant_id -> permet de rattacher un employé ou un client à un restaurant.

Table «role»

role
role_id: TINYINT NOT NULL [PK]
name: VARCHAR(32) NOT NULL

Role:

Permet de définir le rôle d'un utilisateur. Cette table permet de ne pas utiliser de type enum.

Table «app_order»

app_order
order_id: INTEGER NOT NULL [PK]
user_id: INTEGER NOT NULL [FK]
restaurant_id: INTEGER NOT NULL [FK]
address_id: INTEGER NOT NULL [FK]

Order:

Permet d'enregistrer toutes les commandes passées par un client ou un employé.

FK: restaurant_id -> permet de rattacher une commande à un restaurant.

FK: user_id -> permet de rattacher une commande à un utilisateur (client/employé).

FK: address_id -> permet de rattacher une commande à une adresse. Car la commande peut être livrée chez un client ou être à emporter.

Un client peut aussi avoir plusieurs adresses.



Table «order_status»

order_status

order_id: INTEGER NOT NULL [PFK]
created: TIMESTAMP NOT NULL
prepared: TIMESTAMP NOT NULL
shipped: TIMESTAMP NOT NULL
delivered: TIMESTAMP NOT NULL
transaction_id: INTEGER [FK]

Order status:

Permet de stocker le statut d'une commande et savoir a tout instant où celle-ci en est, notamment pour notifier le client et les employés du restaurant.

L'attribut *paid* peut être null, indiquant alors que la commande n'a pas été payée.

PFK: order_id -> rattache une commande à son statut.

*FK: transaction_id -> permet de savoir si la commande a été payée ou non. Si la transaction est **null** alors la commande n'est pas payée.*

Table «order_product»

order_product

product_id: INTEGER NOT NULL [PFK]
order_id: INTEGER NOT NULL [PFK]
quantity: TINYINT NOT NULL

Order product:

Table de jonction entre **order** et **product**.

Permet de savoir quelle commande contient quel produit et quelle quantité de ce produit.

PFK: product_id -> permet d'identifier un produit dans une commande.

PFK: order_id -> permet d'identifier la commande.



Table «address»

address

address_id: INTEGER NOT NULL [PK]
user_id: INTEGER [FK]
supplier_id: INTEGER [FK]
restaurant_id: INTEGER [FK]
title: VARCHAR(128)
street_number: VARCHAR(10) NOT NULL
street_name: VARCHAR(256) NOT NULL
postal_code: VARCHAR(16) NOT NULL
town: VARCHAR(128) NOT NULL
latitude: FLOAT
longitude: FLOAT

Address :

Stock les adresses des utilisateurs (clients/employés) et fournisseurs.

Les attributs latitude et longitude seront remplis via l'API Google Maps qui permet de traduire une adresse en coordonnées GPS.

Ces données permettront alors de proposer au client le restaurant le plus proche et calculer le meilleur itinéraire de livraison.

FK: user_id -> permet d'identifier si l'adresse est celle d'un utilisateur et lequel.

FK: supplier_id -> permet d'identifier si l'adresse est celle d'un fournisseur et lequel.

FK: restaurant_id -> permet d'identifier si l'adresse est celle d'un restaurant et lequel.

Table «supplier»

supplier

supplier_id: INTEGER NOT NULL [PK]
name: VARCHAR(45) NOT NULL
description: VARCHAR(300)
email: VARCHAR(100) NOT NULL
phone_number: VARCHAR(16) NOT NULL

Supplier :

Stock tous les fournisseurs de tous les restaurants.

Table «supplier_ingredient»

supplier_ingredient

supplier_id: INTEGER NOT NULL [PFK]
ingredient_id: INTEGER NOT NULL [PFK]

Supplier ingredient :

Table de jonction entre **supplier** et **ingredient**.

Possibilité d'avoir plusieurs fournisseurs qui fournissent plusieurs ingrédients.

PFK: supplier_id -> identifie un fournisseur.

PFK: ingredient_id -> identifie un ingrédient.



Table «restaurant»

restaurant

restaurant_id: INTEGER NOT NULL [PK]
name: VARCHAR(64) NOT NULL
email: VARCHAR(128)
phone: VARCHAR(16)

Restaurant :

Stock les différents restaurants, leur nom et leurs coordonnées de contact.

Table «stock»

stock

stock_id: INTEGER NOT NULL [PK]
restaurant_id: INTEGER NOT NULL [PFK]
ingredient_id: INTEGER NOT NULL [PFK]
quantity: DECIMAL(10, 2) NOT NULL
change_datetime: TIMESTAMP NOT NULL

Stock :

Permet la gestion des stocks des aliments de chaque restaurant.

Un timestamp permet de savoir quand le stock a été changé (ajout/suppression).

FK: ingredient_id -> permet d'identifier l'ingrédient en stock.
FK: restaurant_id -> permet d'identifier le restaurant auquel est attaché l'ingrédient.

Table «ingredient»

ingredient

ingredient_id: INTEGER NOT NULL [PK]
name: VARCHAR(128) NOT NULL
description: VARCHAR(256)

Ingredient :

Stock tous les ingrédients de tous les restaurants.

FK: suppliers_id -> permet d'identifier le fournisseur habituel de l'ingrédient en question.

Table «product_ingredient»

product_ingredient

product_id: INTEGER NOT NULL [PFK]
ingredient_id: INTEGER NOT NULL [PFK]
quantity: DECIMAL(5, 2) NOT NULL

Product ingredient :

Table de jonction entre **ingredient** et **product**.

Défini la quantité d'ingrédient nécessaire à la réalisation d'un produit (plat).

PFK: ingredient_id -> identifie un ingrédient.
PFK: product_id -> identifie un produit.



Table «product»

product
product_id: INTEGER NOT NULL [PK]
name: VARCHAR(64) NOT NULL
description: VARCHAR(256)
price: DECIMAL(5, 2) NOT NULL
img_filename: BLOB

Product:

Stock tous les produits de tous les restaurants.
Une description et une image sont optionnels.

Le client peut consulter cette fiche et savoir ce qu'il commande.

Le pizzaiolo consulte cette fiche pour se rappeler les ingrédients d'un plat.

Table «product_category»

product_category
product_id: INTEGER NOT NULL [PFK]
category_id: INTEGER NOT NULL [PFK]

Product category:

Table de jonction entre **product** et **category**.
Permet d'identifier plusieurs produits dans plusieurs catégories.

PFK: product_id -> identifie un produit.

PFK: category_id -> identifie une catégorie.

Table «category»

category
category_id: INTEGER NOT NULL [PK]
name: VARCHAR NOT NULL
parent_category_id: INTEGER [FK]

Category:

Stock les différentes catégories dans lesquelles peuvent être classés les produits.

Une catégorie peut avoir une ou plusieurs catégories parentes.

Table «transaction»

transaction
transaction_id: INTEGER NOT NULL [PK]
order_id: INTEGER NOT NULL [FK]
transaction_datetime: TIMESTAMP NOT NULL
refund_cause: VARCHAR(512)

Transaction:

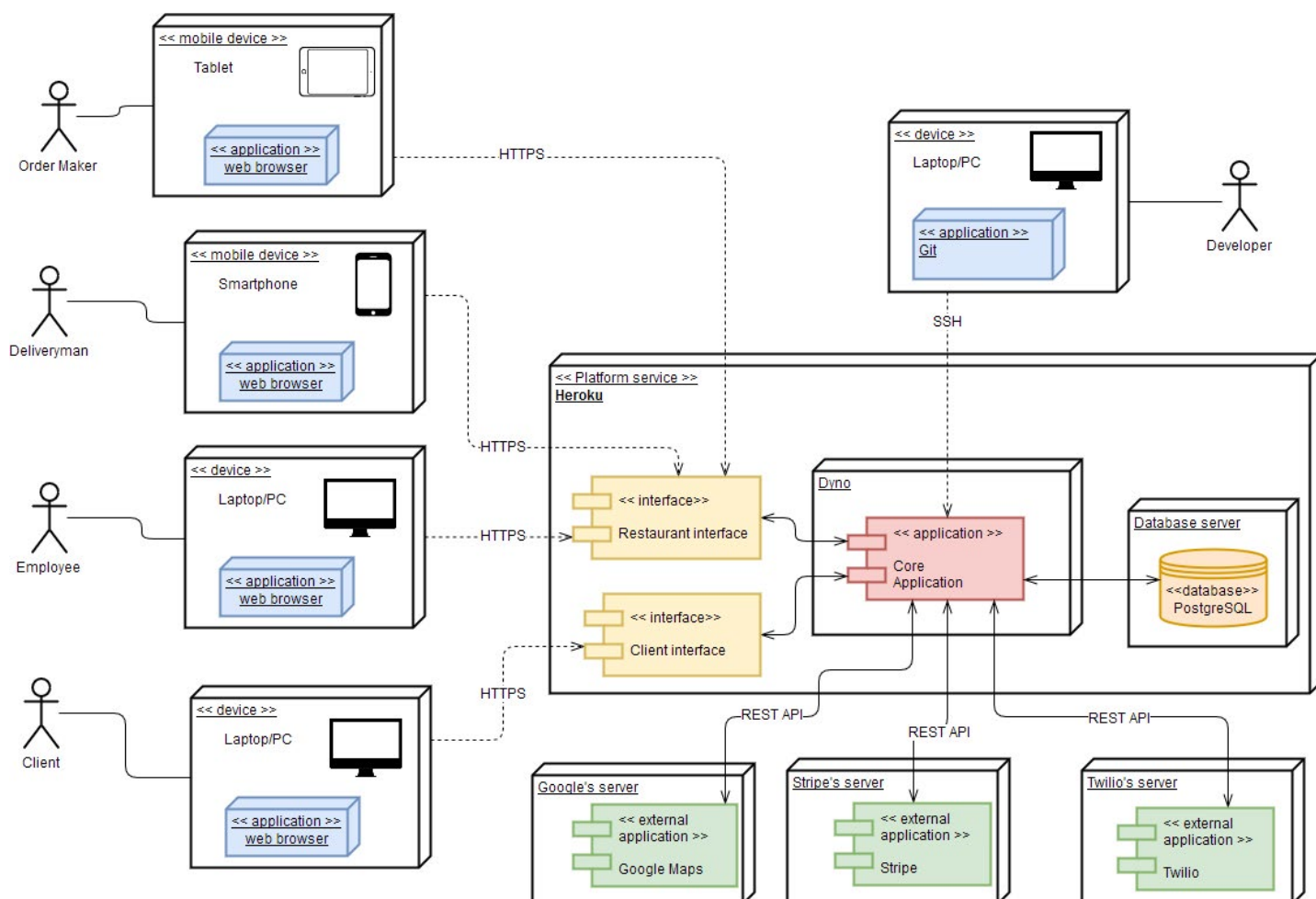
Cette table enregistre les paiements et remboursements effectués auprès des clients.

Un timestamp permet de connaître le moment de paiement ou remboursement d'une commande.

FK: order_id -> permet d'identifier la commande liée à la transaction et donc le prix de celle-ci.



VI. Architecture de déploiement



Le diagramme de déploiement nous permet de comprendre comment sera déployée l'application et par quel moyen chaque acteur devrait y accéder.

Pour une meilleure résolution d'affichage de ce diagramme, un fichier PNG est fourni (OC-Pizza UML deployment.png)

L'application étant déployée sur Heroku, les **développeurs ont simplement besoin d'un terminal** de commande pour déployer celle-ci en production via **git**.

a. Interfaces

Les utilisateurs (employés/clients) accèdent à l'application via leur terminal favori ou le plus indiqué et un navigateur web installé sur celui-ci. La communication se fait via le protocole HTTPS. **Les clients et les employés ont accès à une interface différente.**

b. Terminaux recommandés

Sur le diagramme de déploiement est indiqué le terminal recommandé pour le pizzaiolo (*order maker*) et le livreur (*deliveryman*).



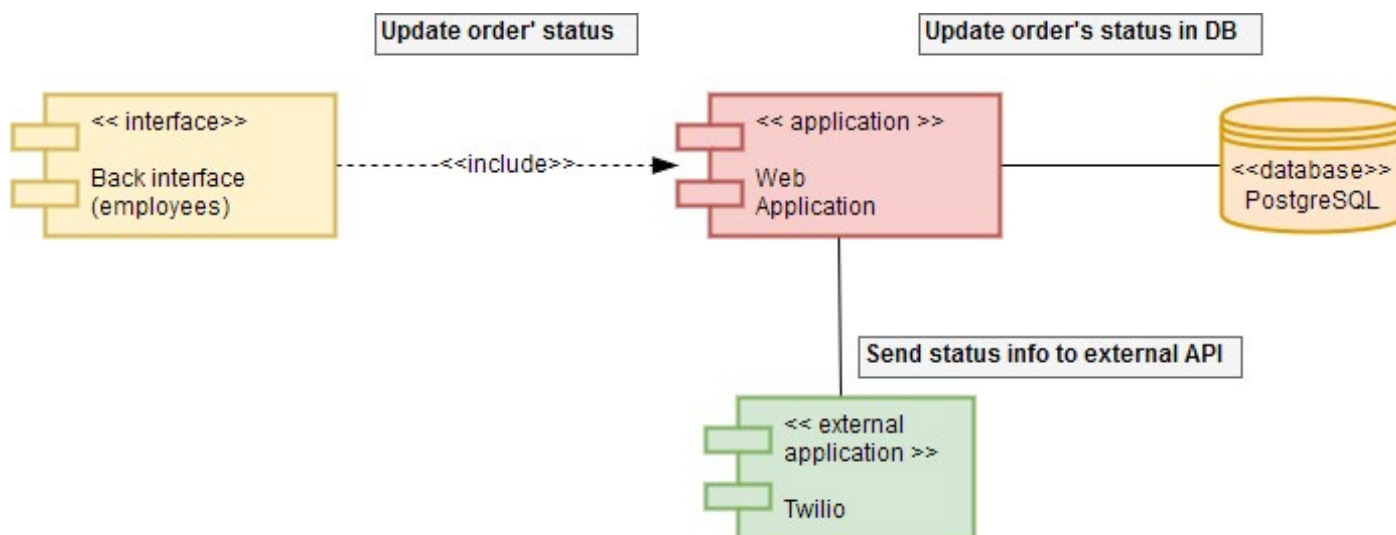
En effet, il serait plus pratique en cuisine d'avoir une tablette qui prend peu de place pour le pizzaiolo. Le livreur lui utiliserait un smartphone afin d'avoir une connexion 3G/4G/GPS et récupérer en temps réel le chemin le plus pratique/rapide pour rejoindre l'adresse du client et le livrer au plus tôt.

c. Application

Le paquet logiciel Python regroupe l'ensemble des fonctionnalités qui effectuent la liaison entre les interfaces et les serveur externes des API et la base de données de l'application.

L'intérêt de cette démarche est de réduire le temps et le coût de développement, d'améliorer la sécurité globale du logiciel et permettre le développement d'interfaces supplémentaires, si nécessaires, en regroupant toutes les fonctionnalités qui peuvent l'être et ainsi obtenir une **application modulaire**.

Exemple d'utilisation du cœur de l'application :



Le cœur permet de faire le lien entre la base de données, les interfaces et les API externes.

d. Serveur de l'application

L'application sera déployée sur Heroku, une plateforme cloud d'application scalable. Cela permettra à OC Pizza de pouvoir payer la solution en fonction de son utilisation et ainsi payer un tarif en fonction de l'affluence sur la plateforme.

e. Serveur de base de données

La base de donnée PostgreSQL sera déployée sur le serveur Heroku via Heroku Postgres. Le choix d'une base de données PostgreSQL est préférable car bien intégrée avec Heroku. De plus PostgreSQL est très puissant et n'est pas détenu par une entité dont on ne connaît pas les motivations futures.



VII. Acteurs externes

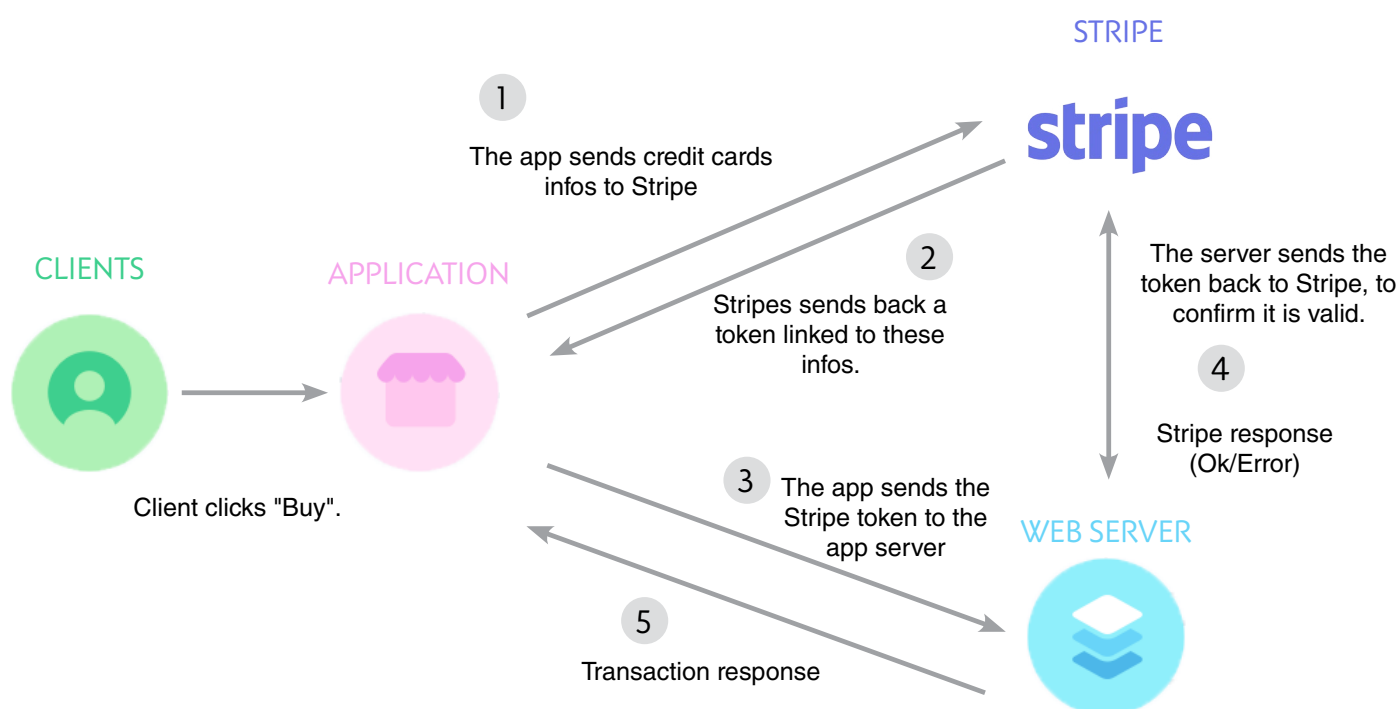
1. Paiements : **stripe**

Gestion des paiements via l'application web.

Très simple à mettre en place, sécurisé et très répandu sur les sites de e-commerce actuels (Deliveroo, Lyft, Instacart, Slack, Ulule...). Documentation très fournie.

Le module **dj-stripe** sera utilisé pour intégrer Stripe à l'application Django créée.

Voici un petit schéma explicatif de comment Stripe fonctionne (inspiré d'un diagramme fourni par Stripe !)



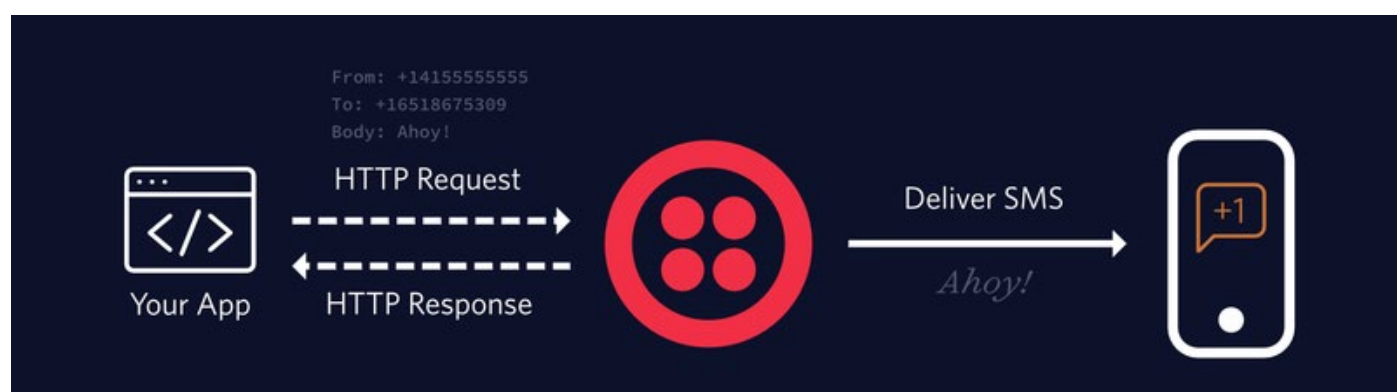


2. Notifications : **twilio**

Permet de notifier l'utilisateur par SMS mais aussi directement dans le navigateur (push).
Scalable, mise en place assez simple et documentation fournie.
Twilio fourni une librairie Python pour intégration.

Le module **django-twilio** sera utilisé pour intégrer Twilio à l'application Django créée.

Voici un simple schéma permettant de comprendre (très) rapidement comment Twilio fonctionnerait avec notre web application Oc Pizza.



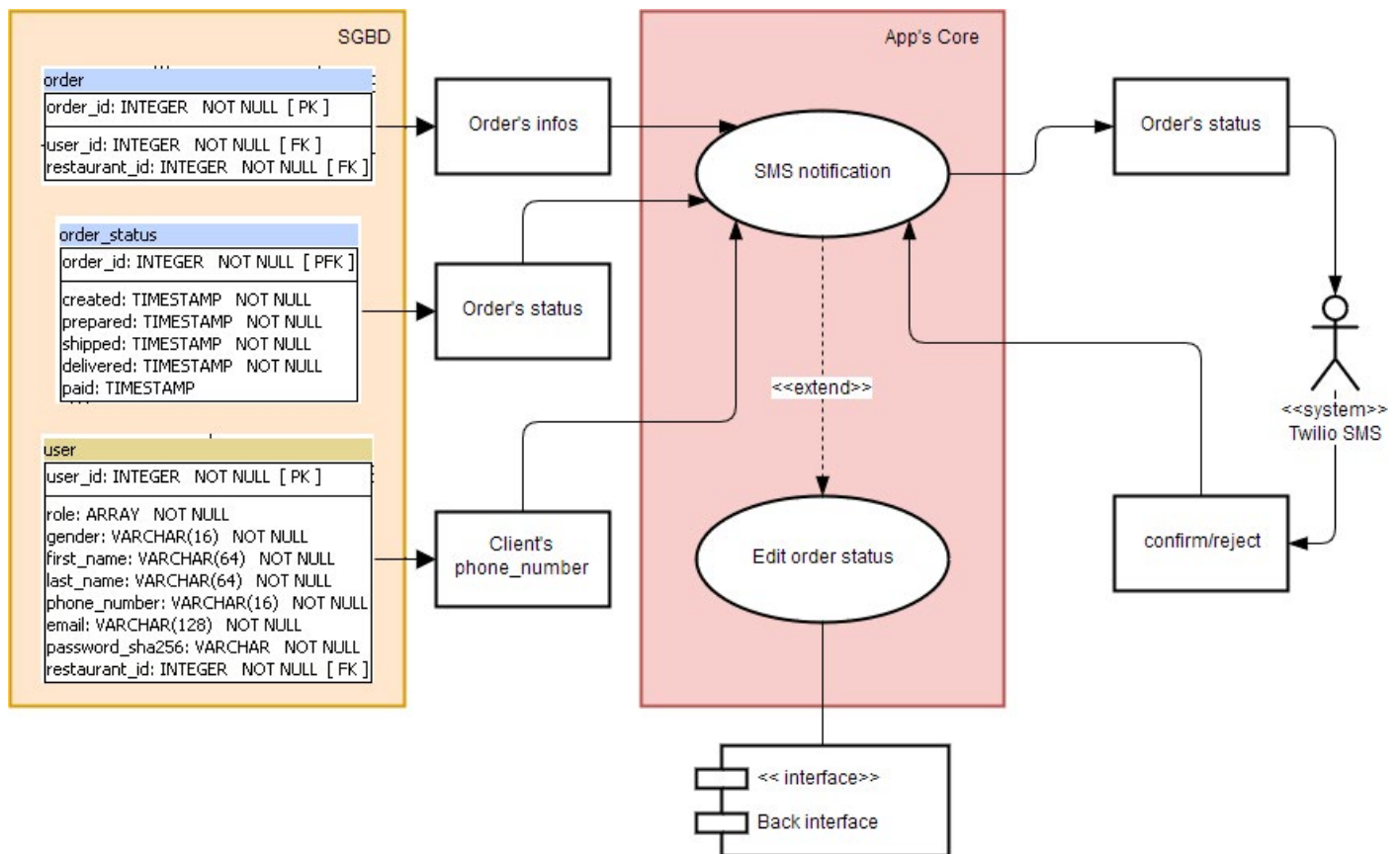
En page suivante, un schéma qui explique plus clairement comment l'implémentation de Twilio se ferait avec l'application.

La base de données communiquerait au cœur de l'application les informations client nécessaire (le numéro de téléphone), la commande associée et son statut actuel.

Ensuite, suivant les paramètres souhaités par OC Pizza, une notification SMS serait envoyée au client pour un/des statut particuliers.

A noter

Cette solution est payante, et cette implémentation reste donc à valider avec le client, mais elle est compatible avec le modèle physique de données.



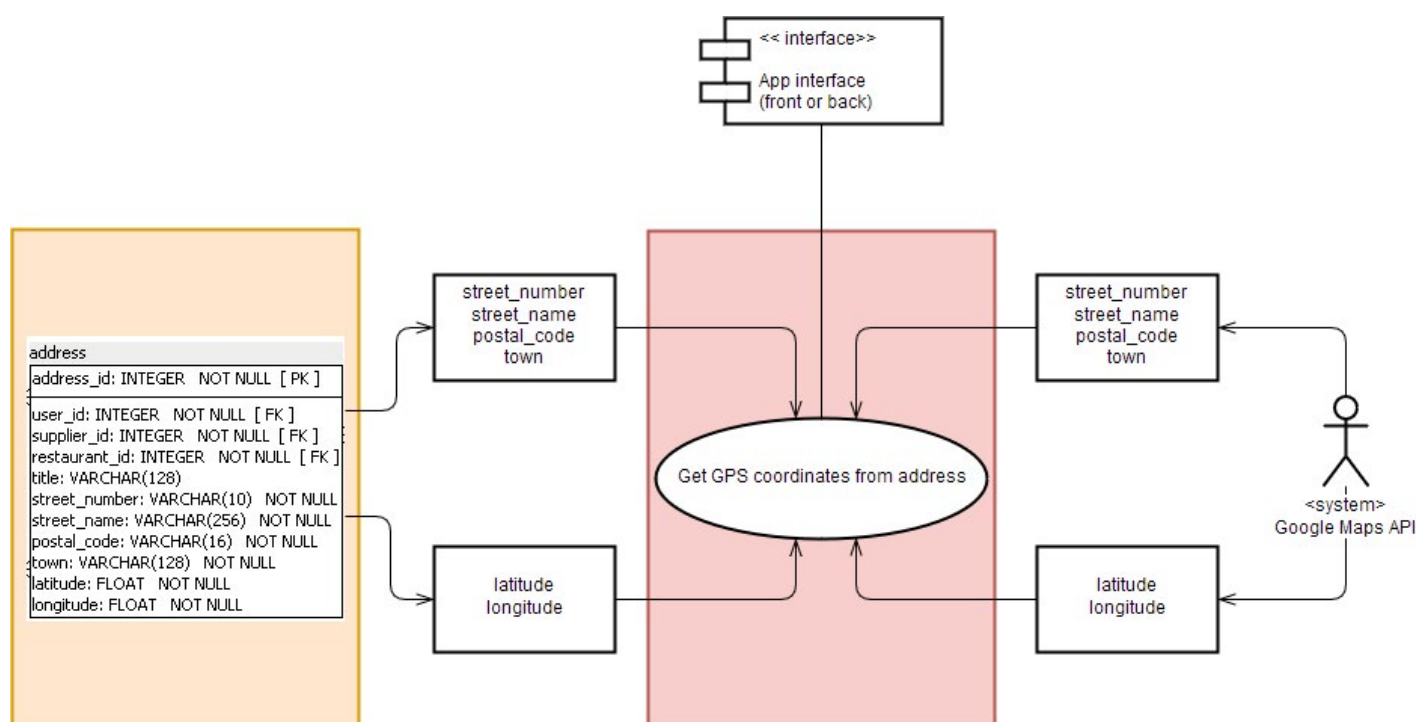


3. Maps : Google Maps

- Affiche la position des restaurants pour l'utilisateur ;
- Permet au livreur de se rendre efficacement chez le client grâce aux informations sur le trafic ou les travaux en cours ;

Il existe une librairie créée par la communauté Python pour utiliser facilement Google Maps au sein d'une application Python/Django. (<https://github.com/googlemaps/google-maps-services-python>)

L'API Google Maps permettra de transformer les adresses des utilisateurs/restaurants en coordonnées GPS.



Ces coordonnées GPS seront ensuite enregistrées sous forme de couple longitude/latitude afin de retrouver plus rapidement l'adresse et ne pas la faire passer par l'API à chaque recherche de restaurant / livraison.

A noter

L'utilisation de cette API peut cependant entraîner un coût supplémentaire, fixé à 0,50\$ par 1000 requêtes au delà des 2500 requêtes gratuites journalières.

OpenStreetMaps n'est pas une alternative viable car leur API restreint le nombre de requête par seconde et risque alors de fortement ralentir notre application.

