VYTAUTAS MAGNUS UNIVERSITY

Faculty of Informatics

DEPARTMENT OF APPLIED INFORMATICS

Daniel Morales Estrella

E-Order system effectiveness for a restaurant

Bachelor Thesis

Informatics Systems study programme, state code 6121BX016

Study field Informatics

Supervisor *lector* Vytautas Barzdaitis

degree, name, surname signature, date

Defended prof.dr Tomas Krilavičius

Dean of Faculty signature, date

Kaunas, 2022

Content

ABSTRACT **8**

Resumen **9**

Introduction **11**

Motivation and objectives **11**

- Motivations 11
- Objectives 12

Analysis **13**

- Competitors 13
- Requirements 17
  - Functional Requirements 17
  - Non-functional Requirements 18
- System Architecture 18
  - Microservices 19
    - Client Microservice 20
    - Restaurant Microservice 21
  - Client side and Server side 22
  - Database Design 25
  - Hardware 26
- Technology stack used for developing 26
  - Electron 26
  - Node.js 27
  - Express 27
  - Axios 27
  - Materializecss 27
  - MongoDB 27
  - Docker 28
- Design 28
  - Use case 28
  - Diagram Clases 29
  - Diagram Sequences UML 30
  - Flow of views 31
- Abstract of analysis 32

Development of the system **32**

- Implementation Client Side 32
  - How works Views 33
  - Use RESTful service 35

Implementation Server Side 36
Microservice Client 36
API RESTful 36
Database operations (CRUD implementation) 37
Dockerfile 41
Microservice Restaurant 42
API RESTful 42
Database Operations 43
Dockerfile 45
Deploy App 45
Results **49**
Providing the testing of the system **49**
Conclusions: **53**
Perspective **54**
References **55**
Appendix 1 **57**

Tables:

Table 1: Analysis competitors
Table 2: Differences in the architecture systems
Table 3: Pros and Cons of Microservice
Table 4: API for Client Microservice
Table 5:API for Restaurant Microservice
Table 6: API RESTful
Table 7: Advantages of docker container

Figures:

Figure 1 Home Instacart
Figure 2 Store Instacart
Figure 3 Basket Instacart
Figure 4 Payment Method Instacart
Figure 5 Home UberEats
Figure 6 Basket UberEats
Figure 7 Different architecture for a system
Figure 8 Client and Server side
Figure 9 Database Design
Figure 10 Use case
Figure 11 Diagram Clases
Figure 12: Diagram Sequence for Restaurant and Employee
Figure 13: Diagram Sequence for Owner
Figure 14: Flow of views
Figure 15: Package JSON
Figure 16: Function Renderer.js
Figure 17: Function2 Renderer.js
Figure: 18 Model Rest Client
Figure 19: Example API RESTful
Figure 20 a: Example API create operation
Figure 20 b: Example API read operation
Figure 20 c: Example API update operation
Figure 20 d: Example API delete operation
Figure 21: Dockerfile Users
Figure 22; API RESTful restaurant
Figure 23: Calls for users microservices
Figure 24: Dockerfile for restaurant image
Figure 25: Docker-compose users and mongo 1 containers
Figure 26: Docker-compose restaurant and mongo 2 containers
Figure 27: Docker-compose Volumes and Networks
Figure 28: Container deployed
Figure 29: Running containers
Figure 30: Users microservice, register && login
Figure 31: Product microservice add type && product
Figure 32: Create edit and cancel order
Figure 33: Order, Show order and update orders

LIST OF ABBREVIATIONS

API	Application Programming Interface
BSON	Binary Javascript Object Notation
CPU	Central processing unit
CRUD	Create, Read, Update and Delete
CSS	Cascading style sheets
Gb	Gigabits
Ghz	Gigahertz
HTML	Hypertext Markup language
JS	Javascript
JSON	JavaScript Object Notation
Mb	Megabits
NoSQL	No Structured Query Language
OS	Operative System
Ram	Random Access Memory
REST	Representational state transfer
UI	User interface
UML	Unified Modeling Language
URL	Uniform Resource Locator

ABSTRACT

Author	Daniel Morales Estrella
Title	E-order System
Supervisor	Vytautas Barzdaitis
Presented at	Vytautas Magnus University, Faculty of Informatics, Kaunas 25-05-2022
Number of pages	54
Number of appendices	9

In this thesis, I develop a web app for restaurants, where customers could order their food without the intervention of a waiter to take the order. Therefore, the app will have 2 different parts, first one for the customers to take the orders and the second for the owner and his employees. This part is for adding the different dishes on the menu and the employees can control the delivery of the orders and charge the food.

During the development of the thesis, we will discuss the different phases of the development of the application. Starting with the theory of how the app works, moreover explanations of which technologies I use for the development, the implementation of those technologies on the application. Finally, I do some tests on the app.

For the implementation of this application, I use electron and node js for the backend and frontend. In addition to storing the data, I use the database MongoDB. Finally, for the deployment of the application, I use docker.

Resumen

Autor	Daniel Morales Estrella
Título	E-order System
Supervisor	Vytautas Barzdaitis
Presentado en	Vytautas Magnus University, Faculty of Informatics, Kaunas 25-05-2022
Número de páginas	54
Número de indices	9

En esta tesis se desarrolla una aplicación web para un restaurante, donde los clientes puedan pedir su comida sin la intervención de un camarero que tome la orden. Por lo tanto, al desarrollar la tenemos dos partes diferentes, la primera es la parte para los clientes, donde estos podrán hacer su pedido y la segunda parte es para el propietario y los empleados. En esta parte se podrán añadir los diferentes platos del menú y los empleados podrán controlar la entrega y cobrar los pedidos.

Durante el desarrollo de la tesis, se explican las diferentes fases del desarrollo de la aplicación. Empezaré con la teoría de cómo funcionará la aplicación, luego sobre cómo se desarrolló esta, seguire con la implementación de las tecnologías para esta aplicación y finalmente hice unas pruebas del sistema

Para la implementación de esta aplicación, he utilizado electron y node.js para el backend y frontend. Además para guardar los datos se utilizó la base de datos de MongoDB. Para finalizar para el despliegue de la aplicación se utilizó docker

1. Introduction

When we go to a restaurant, the employees have many tasks that extend the waiting period between when the customer arrives, order his food and pay for the service. So this makes a worse experience for the customers and increases the time the client spends at the restaurant, then if the restaurant has fewer places available then they have fewer revenues. To solve this problem, what we can do is to reduce those waiting times in which the customer has to wait to be served by the waiter.

Therefore, I do a system to reduce this dead times, if we analyse this part we have three important part to reduce the time: first when the client arrives at restaurant and need to wait the menu, secondly, when he have to be served to take his meal order and third order the bill without having to wait for the waiter.

Then I develop an application that manages these processes.

Then, the task is developing a system to handle the orders on the restaurants, where the owner will need to add his products, handle his employees, also would be great to control the sales through the app

2. Motivation and objectives

2.1. Motivations

The hospitality industry is one of the most important in the Spanish economy. Tourism is one of the cornerstones of the Spanish economy and an outstanding driver of economic and social development. In 2017, it accounted for 11.8% of GDP and in 2018 sustained 13.5% of employment (or 2.6 million direct jobs). (OECD, 2020)[0]

Although this sector needs a lot of people for the work, because there are some tasks where we waste a lot of time (taking orders, inventory control, purchasing food, etc.) . Sometimes this leads to long waiting times, causing customer discomfort. If we automated this kind of task the workers could do other kind of tasks, reducing these waiting times.

It is amazing how this kind of task could be automated but here in Europe still with these old habits. For example, if we compare with the Asian continent there they have some automated work, for example robots instead of waiters.

2.2. Objectives

The objective for the system in this thesis is to develop a system to take the orders in a simple way and take care of the whole logistic process, to facilitating the workflow of the restaurant.

Also, this project is good to learn about the different phases in the development of an app, put in practice the different technologies I studied during my time as student, moreover to improve my knowledge about these technologies.

To implement the objective small tasks will be solved:

- Overview of similar systems
- State the functional and non-functional requirements
- Choose the system architecture for the system
- Describe the frontend, backend and database
- Explain how the system works
- Construct frontend
- Develop a backend
    - Authentication
    - Multiuser system
    - Connection with database, CRUD implementation
    - Client side:
        - Register
        - Login
        - Update user profile
        - Order food
        - Update order
        - Payment method
    - Restaurant microservice
        - Create employees
        - Create menu
        - Sales control
        - Sales statistics
        - Kitchen control: handle products and orders,
        - Dining room control
- Testing the system

3.Analysis

3.1. Competitors

It is important to analyze the market before making a new application, for this reason I take 2 companies specialized in the delivery of food and other orders, as I did not find any company focused on taking the orders in a restaurant, but the final method to order is similar.

Compared my app with other possible competitors:

- ➔ Uber-Eats [1] is a branch of Uber company, this branch is specialized a take away food
- ➔ Instacart [2] this company is specialize in groceries and home essentials delivered from local stores

Table1: Analysis competitors

	E-Order System	Uber-Eats	Instacart
Web Pages	System develop in this thesis	https://www.ubereats.com/es	https://www.instacart.com/
Technologies/ frameworks	Electrón Node MongoDB Docker	React [3] Node Nginx Mysql PostgreSQL MongoDB	React [4] Python PostgreSQL Nginx Firebase
Specification function for clients	Login Charge Menú Send order Checkout	Main Page with Restaurants Menu in specific restaurant Payment Method Send order	Main page with shops Add element to the basket Login Checkout
Problems	It is only for restaurant	Not probems	Must be registered for order

After analysing our competitor we can see none of these, manage the internal management in the restaurant, only want to make more sales for them, then we have a place on the market where nobody is taking profit. But one of the risks is that these companies could make something similar to this app, and take advantage of their position in the market. But the problem they have is that this kind of business has some legal problems with the current law with the deliveries, thanks to that they are focusing their resources to solve this problem.

Spain became the first European Union member to give delivery workers labor rights, firms such as Uber Eats or Just Eat are still struggling to adapt to a new law that may become a model for the rest of Europe. (BY FRENCH PRESS AGENCY - AFP, 2021) [5]

Otherwise, in this app we had the problem we need to the restaurant accept use this system, which could be a barrier to entry into the market

Instacart UI

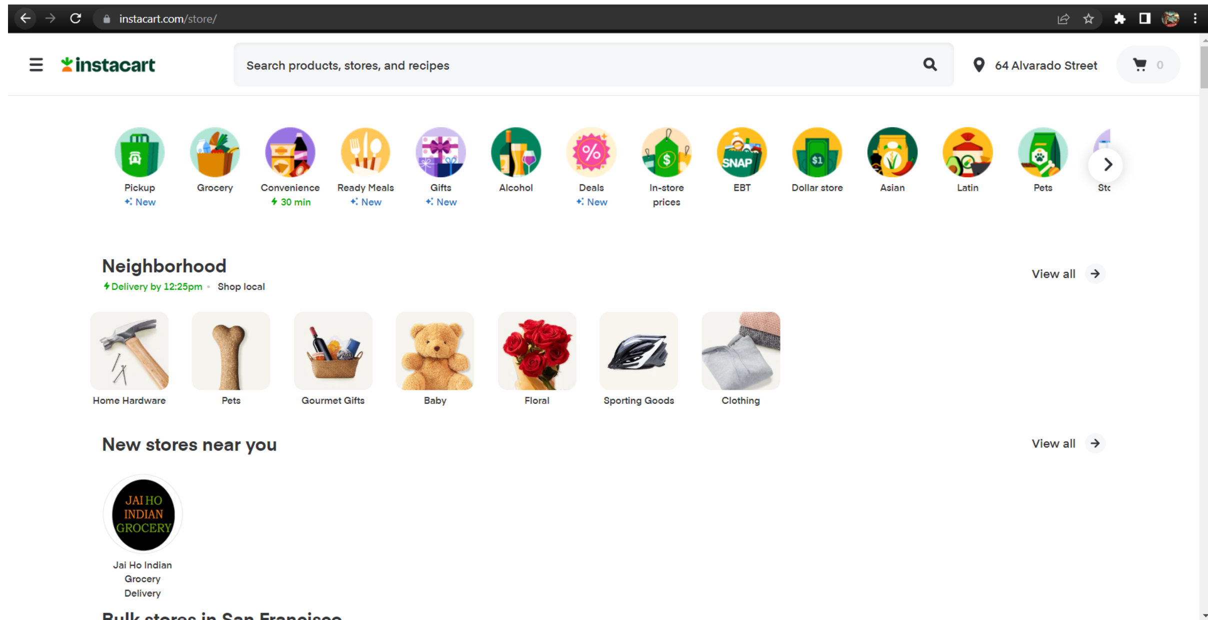


Figure 1. Home Instacart. [https://www.instacart.com/store]

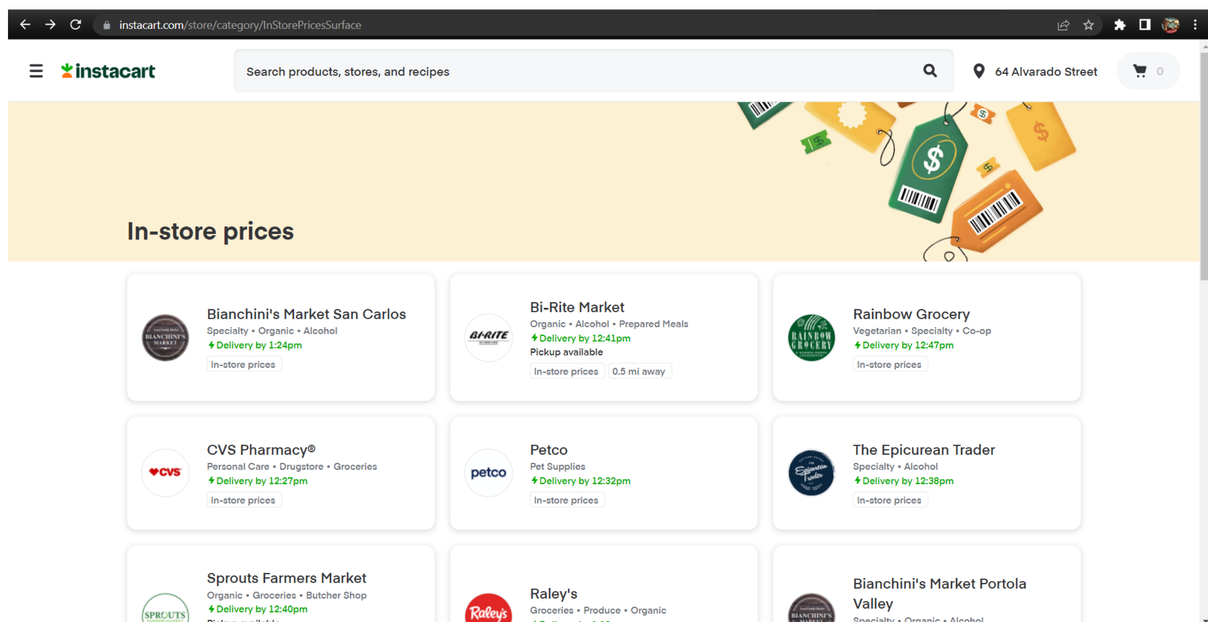


Figure 2. Stores Instacart. [https://www.instacart.com/store/category/InStorePricesSurface]

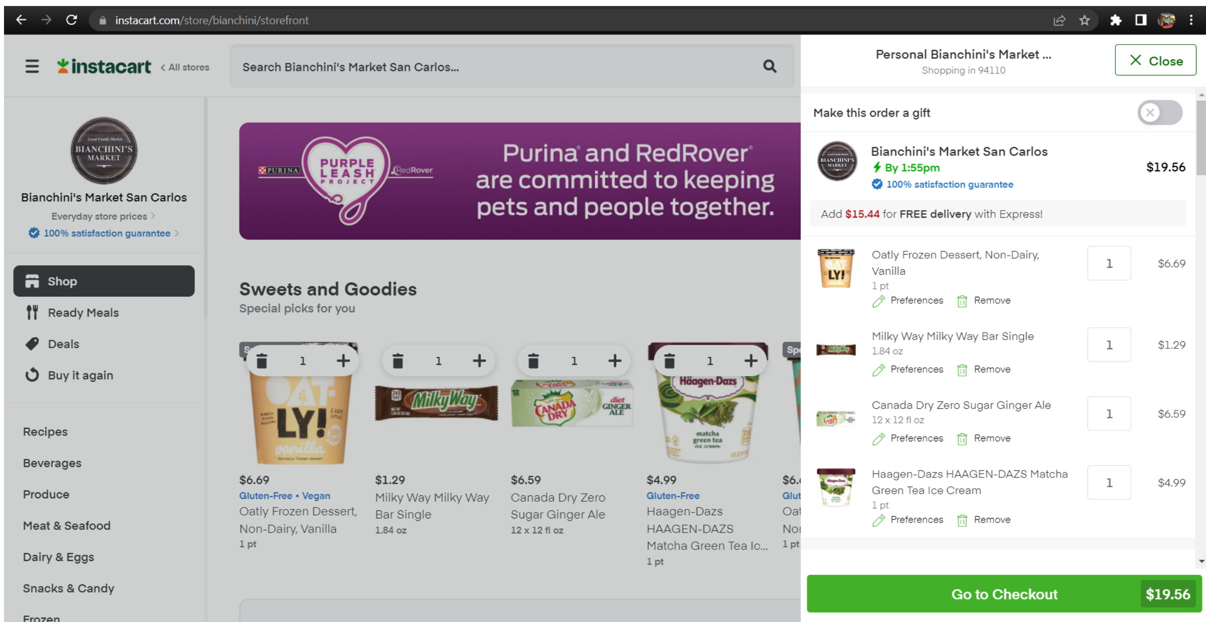


Figure 3. Basket Instacart. [https://www.instacart.com/store/bianchini/storefront]

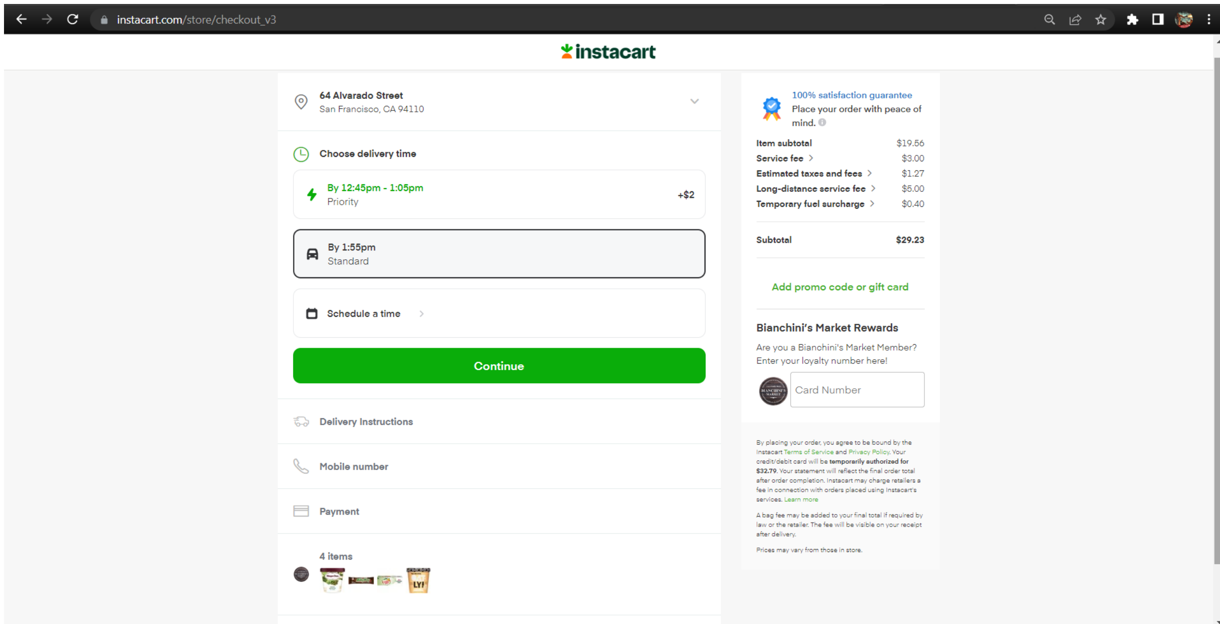


Figure 4. Payment method Instacart. [https://www.instacart.com/store/checkout_v3]

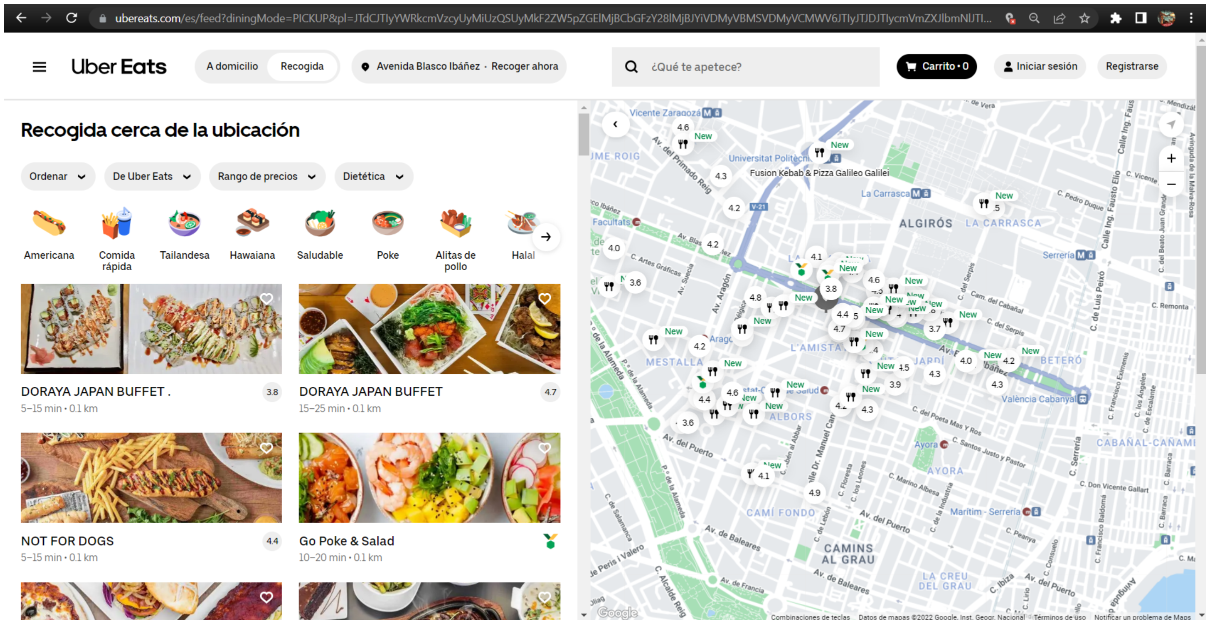


Figure 5. Home Ubereats. [shorturl.at/gnELT]

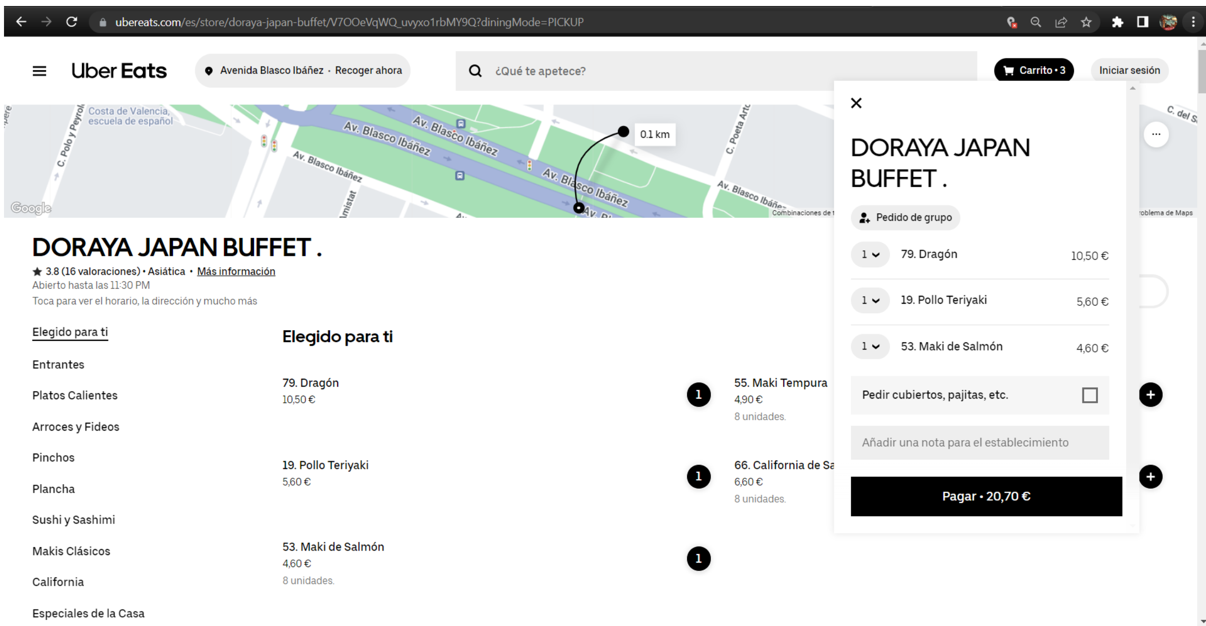


Figure 6. Home Ubereats. [shorturl.at/klyIR]

3.2. Requirements

The first step for develop an app it is to specify which specifications we need for our system works, then we need to choose our functional and non-functional requirements

3.2.1. Functional Requirements

For the functional requirements we need to define our main function in the app, in our structure we can differentiate two important parts, for the clients and for the restaurant. So I define the functional requirements for each part

- Client
  - Register
  - Login
  - Update profile
  - Order food
  - Update order
  - Payment method
- Restaurant
  - Create employees
  - Create menu
  - Sales control
  - Kitchen control
  - Dining room control

3.2.2. Non-functional Requirements

For the non-functional requirement I explain the behaviour of the app, the non-functional requirements are split in three blocks: product requirements, here we specify dependencies on the app works properly, organization requirements, here we know the process for the client and the developers and finally the outside requirements, here we detect the requirements outside the system and his development

- Product
  - The minimal requirements to run the app is: Intel Core i5 2.3Ghz for the CPU, 8Gb for the ram and 500Mb for store
  - App run on Windows and Linux

- Organization
  - App will be developed with electron[6]

- Outside requirements
    - The client side must be connected with the microservices to do the different operations
    - The users data will be attached to the data protection regulation to the home country

3.3. System Architecture

In this section I describe basic components on the app and how they work together, also which is the structure for the database and finally the hardware requirements for run the app

Before to take the decision how the system it is develop, I show the other ways to build systems:

- Monolithic: It is when all the system is develop in ”one piece”
- SOA: It is when the system is develop in different components to provide the functionality to the system
- Microservices: It is when the system is develop in independent services and each service work separately

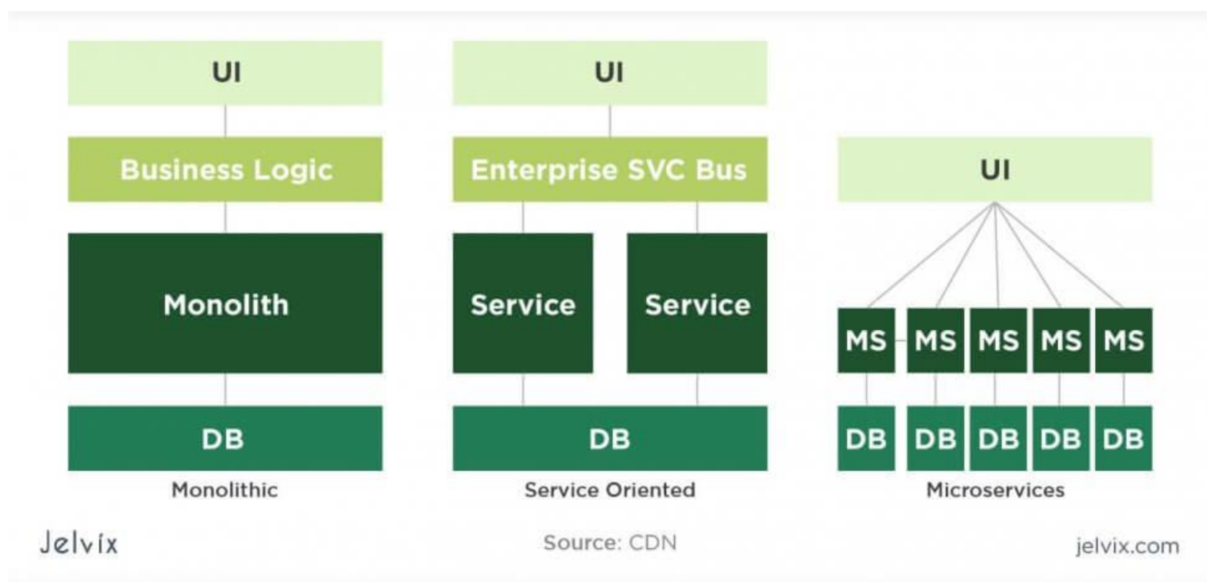


Figure 7 Different architecture for a system [https://jelvix.com/blog/monolith-vs-microservices-architecture]

Table2: Differences on the architecture system

	Monolithic	SOA	Microservice
Design	The design is in one piece	Design a different service, where the communication is by a protocol over a network	Design for small and independent services, where the communication is by an API
Scalability	Complicated for how was developed	Dependencies between services and the network protocol could produce bottleneck	It more easy because each service is independent
Development	All the system required to be developed with the same stack	Could reusable components and services	Each service could be developed with the most appropriate technology for its function

3.3.1. Microservices

Microservices are when we split our functionalities of app, in small independent services to build a distributed system where they work together. The reason for this is to make it easier to scale or update the services without disrupting other services in the app.

Microservice are the best way to develop the system due to "now are a new trend in software architecture, which emphases the design and development of highly maintainable and scalable software" (Springer, 2017)[7], also if I would develop the system with the monolithic architecture we would have problems like:

- Less scalability
- Less reusability
- Large code, who made problems at the moment to fix some bugs or understand the system

Table3: Pros and Cons of Microservice

Pros of use Microservices	Cons of use Microservices
More tolerant to errors	More latency
Individual service can scale and easily deployed	Dependencies between microservices
Each service could have different stacks	More difficult to debug
Easy maintenance	More complicated design this architecture

The microservices created for the system are following:

1. Client Microservice

This microservice is for the management of our clients, register, and authentication. This microservice must show the products in the menu to the customers, therefore this microservice is required to establish a communication with restaurant microservice. Then this microservice will do:

- Register users
- Authentication
- Show the menu
- Order food
- Payment

Also it is important to show the API for establish the communication with the database, then here is the app for client microservice:

Table 4: Api for client Microservice

Function	Parameters	Description
addClient(client, cb)	• client: The client • cb(err, client): Callback	Add new clients Callback receive the client
login(nickname, password, cb)	• nickname: Nick for the client • password: Password for the client • cb(err, client, token): Callback	Authentication for the clients Callback receive token, which make easy to identify on the database and do the different functions
getCollection(token, id, cb)	• token: To identify the client • id: id from collection we want • cb(err, collection)	For taking collection from users database Callback receive the collection
updateClient(token, client, cb)	• token: To Identify the client • client: Data to update • cb(error, client)	Update data client Callback receive the new data client
listMenu(token, cb)	• token: To Identify the client • cb(error, data)	List all the product Callback receive the data
createOrder(token, order, cb)	• token: To identify the client who orders • Order: Product ordered by the client • cb(error, food)	Send the order to the kitchen Callback receive dishes for the meal
editOrder(token, orderID, cb)	• token: To identify the client who orders • orderID: To identify the order which it is modify • cb(error, food)	The client can modify his order Callback receive the new order

printTicket(token, userID, cb)	• token • userID: To identify the user • cb(error, ticket)	Print the ticker for the meal Callback receive ticket
cancelOrder(token, orderID, cb)	• token: To Identify the client • orderID: To identify the order • cb(error)	Cancel the current order Callback is for handle errors
removeProductOrder(token, orderID, id, cb)	• token: To Identify the client • orderID: To identify the order • id: To identify the product • cb(error)	Remove specific products in the current order Callback is for handle errors

2. Restaurant Microservice

This microservice is for the owner, where it is possible to manage the different products we have in the restaurant, also it implements a function to check our sales statistics to know which products have better receptions between the customers. Finally, here it is important to implement a special kind of user for the workers in the restaurant (waiters, cooks). Also, it is important to mention that this microservice will be depending on the client microservice because it would need access to the orders stored on the other microservice to list the product from a specific order.

Then the function we will implement here:

- Sales statistics
- Handle product
- Orders

Also it is important to show the API for establish the communication with the database, then here is the app for restaurant microservice:

Table 5: API for Restaurant microservice

Function	**Parameters**	**Description**
addProduct(token, product, cb)	• token: To Identify the employee • product: The product • cb(err, product): Callback	Add new product Callback receive the product
updateProduct(token,productID, product, cb)	• token: To Identify the employee • productID: To identify the produ • product: Changes on the product • cb(err, product)	Modify our current products Callback receive the new product
deleteProduct(token, product, cb)	• token: To Identify the employee • product: product to be deleted • cb(err, product)	Delete products
updateSales(token, orderID, cb)	• token: To Identify the employee • orderID: order with all t products to update	Keep updated our sales Callback is for handle the error

	• cb(err)	
addType(token, type, cb)	• token: To Identify the employee • type: Type for products • cb(err, type)	Create type for identify the products Callback receive the new type
updateType(token, oldtype,type,cb)	• token: To Identify the employee • oldtype: type we change • type: new type • cb(err, type)	Update the new content for the type Callback receive update for the type
deleteType(token, type,cb)	• token: To Identify the employee • type: type deleted • cb(err)	Delete a specific type Callback for handle errors
showOrder(token, orderID, cb)	• token: To Identify the employee • orderID: Order identifier	Show the product in the order Callback receive the order
closeOrder(token, orderID, cb)	• token: To Identify the employee • orderID: id for identify order	Close the order Callback is for handle errors

3.3.2. Client side and Server side

At the moment to develop the system it is important to know how the system will work, for this I will split in two important sides:

- Client side: This side of the system contain the interfaces, also from this side we need to create a file to consume the RESTful API service implemented on the server side

- Server side: This side of the system contains the 2 microservices, which implement the RESTful API services listening on different ports to call the database, to insert the data from the client. Although it is important to advise between our microservices have 1 dependency as we can see on e.g figure 8 the microservice restaurant require some information of the clients microservice

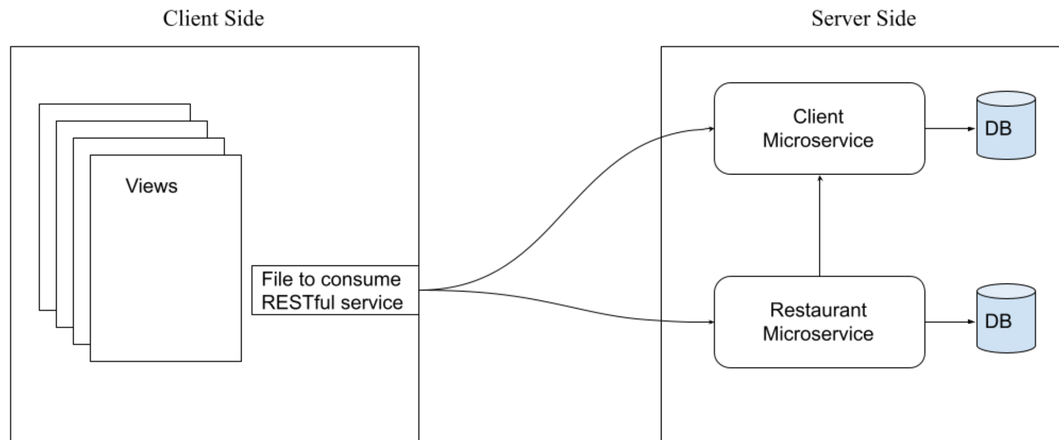


Figure 8: Client and Server Side made by the author

Moreover, as we mentioned before, on the server side a RESTful API service, which will be consumed for the client side. Then for developing our RESTful API service it is important to focus on the most important elements in the system, then for this system it is the customers, product and order.

So we need to define the main entry points into the system. The integration with E-Order System API will be implemented:

- /e-order/customers
- /e-order/product
- /e-order/order

Although it is important to define some special entry points for the system

- /e-order/sessions
- /e-order/info

This is because when we create a new session, we will need to assign the token to the user who has been logged in and to establish communication between our microservices.

Now I will show a table with the API for our RESTful service:

Table 6 : API RESTful [1]

Method	URL	Description
	Special Resource	
Post	/e-order/sessions	Open new session Data: {email, password} Results: {token, user}
Get	/e-order/sessions	Check the current session Params: token Results: {token, user}
Get	/e-order/info/:id	Get collection from users microservice Data: {id} Results: product
	Users Resource	
Post	/e-order/users	Add new user Data: user Results: user
Put	/e-order/users/:id	Update user with the id :id Params: token Data: user Results: user
	Orders Resource	
Get	/e-order/order	List products and type available Params: token Results: products an types
Get	/e-order/order/open	Show open orders Params: token Results: Open orders
Get	/e-order/order/product/:id	Show products on the order to the user identify with :id Params: token Results: Products on the order
Post	/e-order/order	Create a new order Params: token Data: order Results: order
Put	/e-order/order/:id	Update the order with the id :id Params:token Data: editOrder
Put	/e-order/order/sales/:id	Update field sale on the products in the order Params: token

[1] Table 6 Continue in the next page

Method	URL	Description
Put	/e-order/order/close/:id	Close the order identify with :id Params: token
Put	/e-order/order/:order/:product	Delete specific product from the order Params: token Data: orderID, productID
Delete	/e-order/order/:id	Delete order identify with :id Params: token Data: orderID
	Products Resource	
Post	/e-order/product/type	Add new type Params: token Data: type Results:new type
Post	/e-order/product	Add new product Params: token Data: product Results: new product
Put	/e-order/product/type/:id	Update type with the id :id Params: token Data: updated type
Put	/e-order/product/:id	Update product with the id :id Params: token Data: updated product
Delete	/e-order/product/type/:id	Delete type with the id :id Params: token
Delete	/e-order/product/:id	Delete product with the id :id Params: token

3.3.3. Database Design

In this section I will analyse the structure of the database, it is important to remember we have 2 microservices with their own database, but we have 1 dependency between them. To implement the database I used MongoDB. The design of DB is presented in e.g.figure 9.

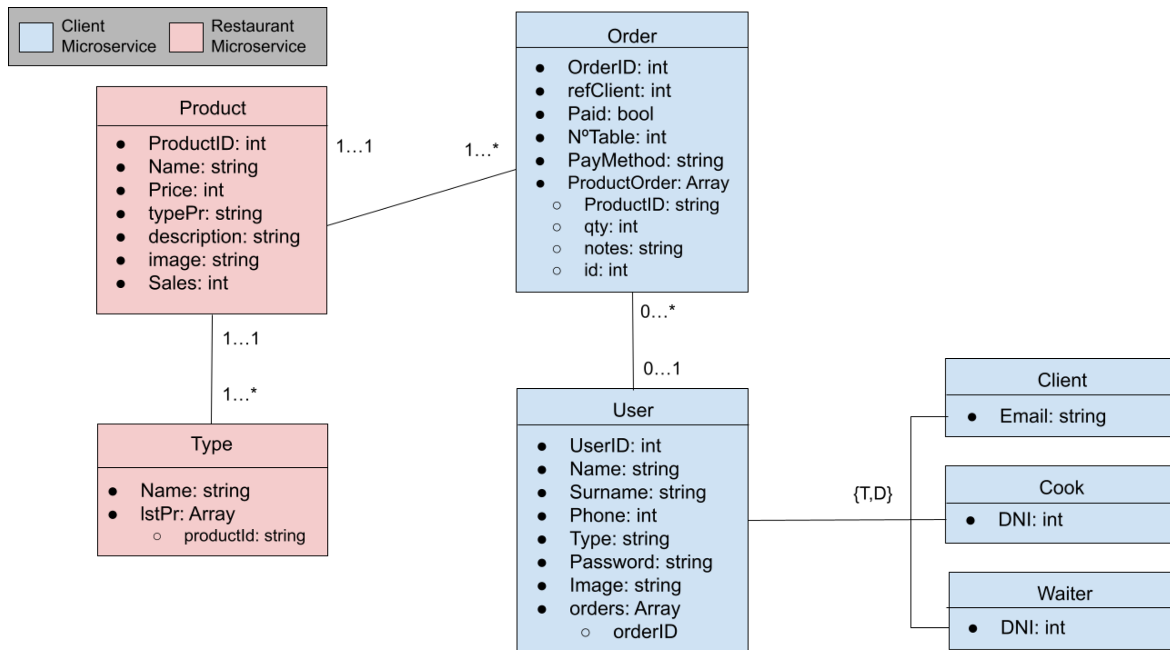


Figure 9: Database Design made by the author

3.3.4. Hardware

To develop a new system, it is important to know the hardware specifications I have. Then for development this system I have a laptop, which the following specifications:

- OS: Windows 10
- CPU: Intel Core i5-6300HQ 2.30GHz
- Ram: 8GB
- Store: SSD 1 Tb and HDD 500 Gb

3.4. Technology stack used for developing

In this section I talk about the technologies I use for the development of the system

3.4.1. Electron

Electron [6] is a framework for creating native applications with web technologies, because electron have two important factors, one is chromium for building interfaces due to chromium being able to interpret HTML, CSS and JS and secondly node.js which we will explain later

I use electron to build our client side, so here I build our interfaces for the customers and the restaurant, also need to implement the RESTful service to call the microservices

3.4.2. Node.js

Node/.js [8] is a cross-platform, open source, runtime environment based on JS. It was created with the objective to be useful in the creation of highly scalable network programmes, such as web servers. This technology that specializes in creating highly efficient applications, due to two fundamental characteristics:

- There is only one thread of execution.
- All input/output operations with asynchronous

For this reasons Node.js can handle thousands of requests, which I require for deploy the microservices

At the beginning was made to run on servers but with other technologies as Electron we can make a full application

3.4.3. Express

Express [9] is one of the most important frameworks in node.js because it provides a robust set of features to develop web and mobile applications. Although I use it to provide server-side logic because one functionality of express is to handle the http request and connect to the operation we require.

3.4.4. Axios

Axios [10] is a promise based HTTP client for the browser and Node. js. Axios makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations.
For that reason, I use axios on the client side to send our request to the microservices.

3.4.5. Materializecss

Materializecss [11] is a CSS and JS framework based on Material Design by Google. It was developed for the front-end, we have multiple UI components to build our app.

I take profit from some components of materialize for improving the UI in the client side.

3.4.6. MongoDB

MongoDB [12] is a NoSQL database oriented to collections, the advantages of this database are its high scalability and flexibility. Another important point on this database is instead of saving data on tables like a relational database, MongoDB uses BSON structures, making better integration with the data, thanks for that this helps to have better performance.

I need two different MongoDB databases, one for each microservice I use

3.4.7. Docker

Docker[13] is a technology which makes it easy to deploy an app inside a container. Although docker adds a new layer over a Linux command to isolate different procedures and resources. Then docker creates an isolated container to deploy the app.

I used docker for containerizing each microservice because it is the most popular option to develop microservices: “Docker containers are becoming an attractive implementation choice for next-generation microservices-based applications. When provisioning such an application, container (microservice) instances need to be created from individual container images“ (Nathan et al., 2017)[14]

Container in docker is an isolated process which has its own memory, network and process. These containers are advantageous due to these containers consuming fewer resources than virtual machines. To create a new docker container it is required to have an image, which has the information to deploy the system (files, libraries, configuration, etc.). We could find all the images from docker in docker hub[15] where we have around more than 100.000 images from software vendors, open-source projects, and the community. Then I use containers to wrap the microservices and provide independence for our app and have better scalability

Instead of use docker I could use a virtual machines but this have more disadvantages than docker:

Docker Containers	Virtual Machines
Every container runs with the kernel of the host	The virtual machines have his own os
Few resources to create a container	More resources than docker containers
Faster to startup the services	Slower top startup the services
Less memory is require	More memory is required

Table 7: Advantages of docker containers

3.5. Design

In this section, I describe how the system works, the different interaction between the views and how the user interact with the system

3.5.1. Use case

With the use case, we represent how the users can interact with the function requirements (point 3.21) specified before. As we can see on the e.g. figure 10, we have three different types of users, which have specific functions depending on them. It is important to remark that we differentiate the user with a tag at the moment to create the users. Also, for the customer, we have an extra option in make order for editing the current order

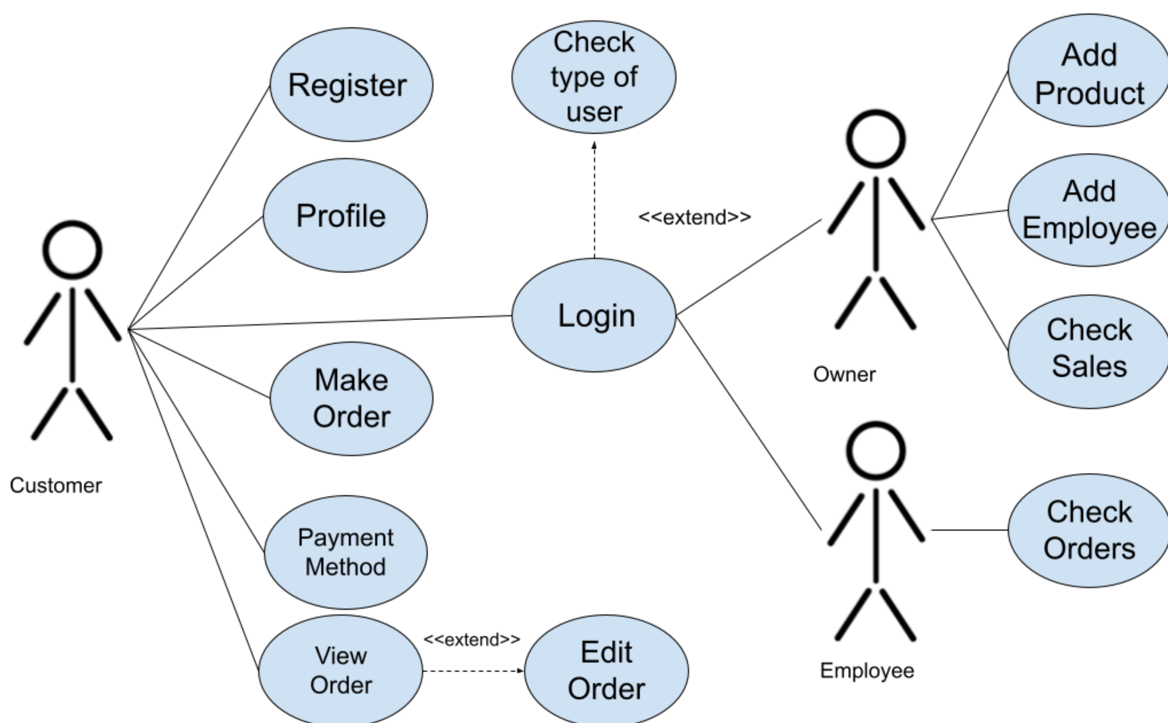


Figure 10: Use case diagram made by the author

3.5.2. Diagram Clases

In this section we analyze the different object we have in the system, we can see in the e.g. figure 11 we have the main classes with his different attributes and the main functions that they have

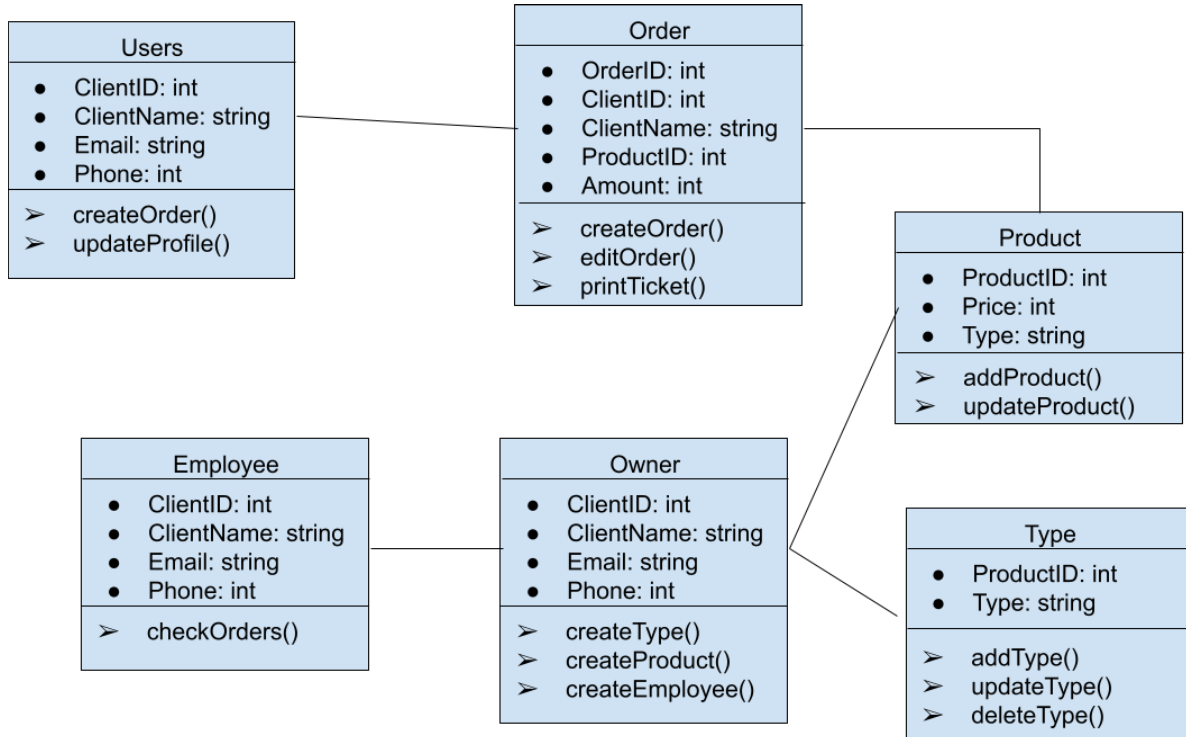


Figure 11: Diagram Clases made by the author

3.5.3. Diagram Sequences UML

In this section I show how our customers interact with the system, for this reason I create two different diagrams, e.g. figure 12 it is how the clients and employees interact with the system, in other hand in e.g., figure 13 it is how the owner interact with the system

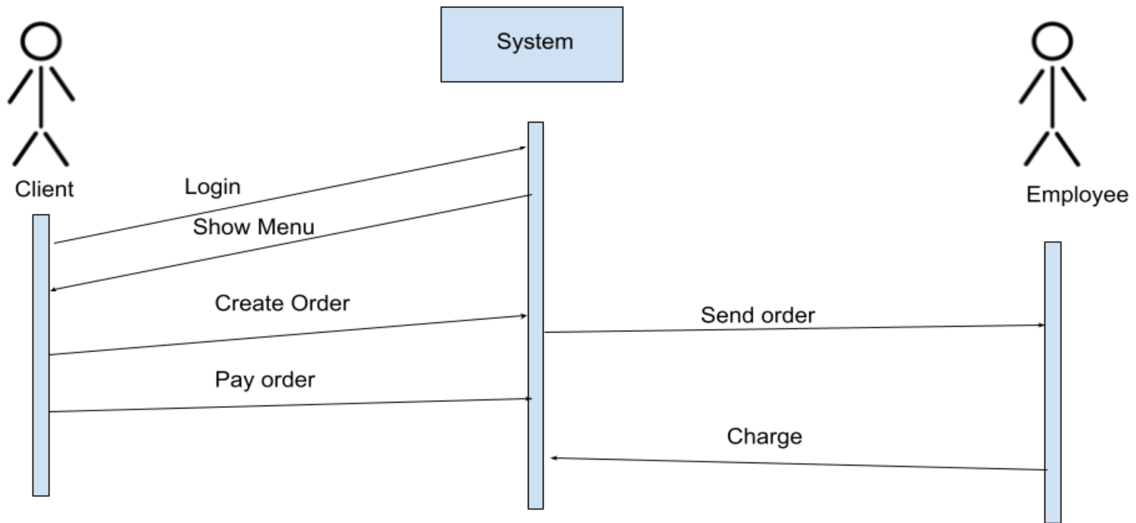


Figure 12: Diagram Sequence for Clients and Employees made by the author

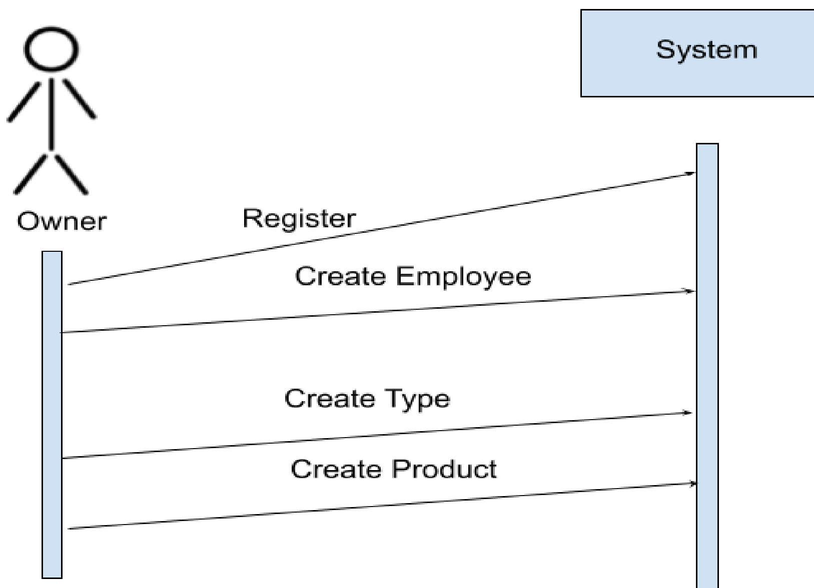


Figure 13: Diagram Sequence for Owner made by the author

3.5.4. Flow of views

Here I show how it is possible to navigate between the different views in this system. In the e.g figure 14 I have two dotted arrows. That means when we create an order this change produce a change in the kitchen view, similar with ticket and dining room when the ticket it is ordered this action produce a change on dining room view.

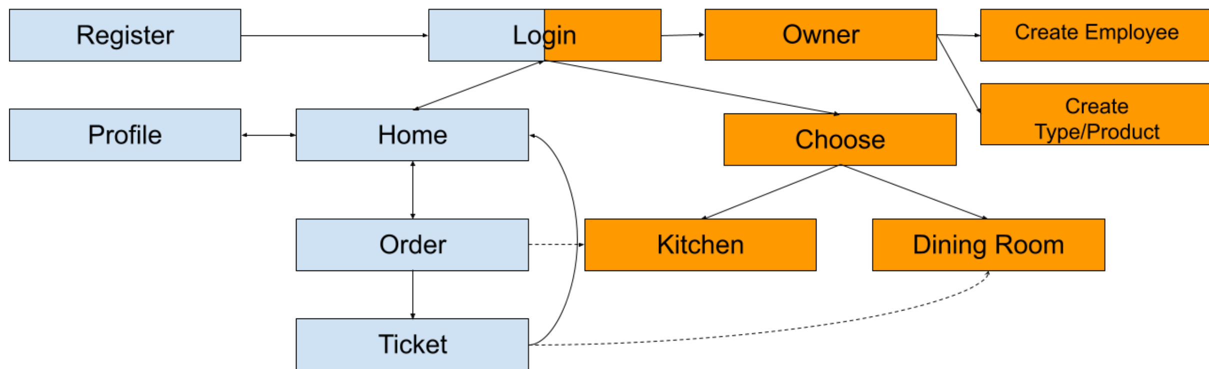


Figure14: Flow of views made by th author

3.6. Abstract of analysis

After analysing possible competitors, I could subtract the basic elements of how it will work and how we will build the system. I have also explained a bit about the technologies that used during the development. On the other hand, we have analysed how the backend of the application will be organized, and finally we analysed how the frontend will be organized and how it will interact with the user.

4. Development of the system

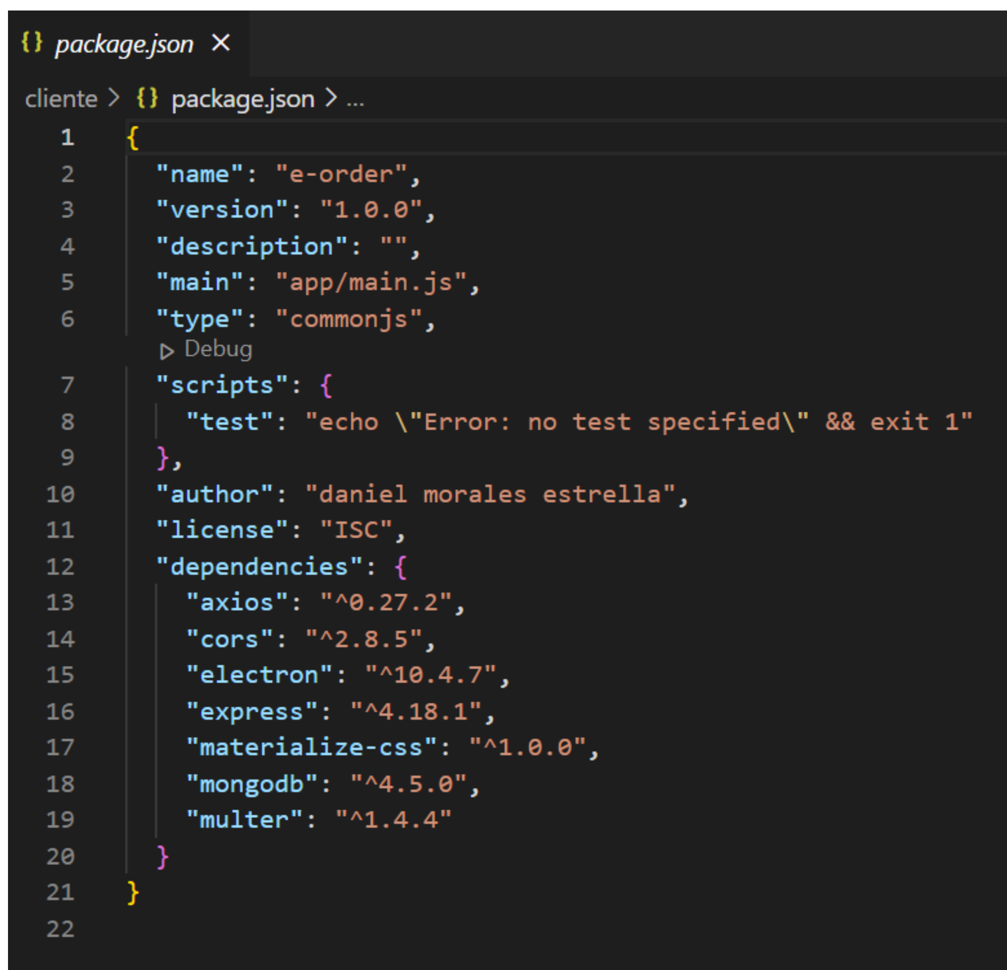
After the analysis of how the system is, it is time to explain how It is developed. I explain a section, starting for the client side, server side and ending with how to deploy the system. Also I explain the most important operations in the system

4.1. Implementation Client Side

Here it explains how the client side was developed and how it works. On this side we have the views, how I use the restful service and how run the client. The internal structure on the client it is:

```
Client/
        package.json
        app/
                | - main.js
                |  - www/
                        | - css
                                | - styles.css
                        | - images/
                                | - products/
                                | - users/
                        | - js/
                                | - model_rest.js
                                | - renderer.js
                        | - views.html
```

Package.JSON is a file with the basic configuration, dependencies, name, where the application starts to run, etc.



```
{
  "name": "e-order",
  "version": "1.0.0",
  "description": "",
  "main": "app/main.js",
  "type": "commonjs",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "daniel morales estrella",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.27.2",
    "cors": "^2.8.5",
    "electron": "^10.4.7",
    "express": "^4.18.1",
    "materialize-css": "^1.0.0",
    "mongodb": "^4.5.0",
    "multer": "^1.4.4"
  }
}
```

Figure 15: Package JSON made by the author

4.1.1. How works Views

After implement the views show on e.g. figure 13, it is important to know how this works, for the development of the GUI I decide to apply the SPA, that's means all the views are indexed on one document, but this views will be hide until the system calls them to be showed, for this reason I develop renderer.js, from this document we ha a collection of all the views, where we going to define the method to show and other function for the system works properly.

Now I explain the main functionalities from renderer.js

```
cliente > app > www > js > JS renderer.js > ...
10  pages.login = {
11      show: function (data) { console.log('login.show()'); },
12      hide: function () { console.log('login.hide()'); },
13      register: function () {
14          console.log('login.register()');
15          navigateTo('register');
16      },
17      login: function () {
18          console.log('login.login()');
19          //Obtenemos las variables del HTML
20          let name = document.getElementById('name').value;
21          let password = document.getElementById('password').value;
22          login(name, password, (err, token, user) => {
23              if (err) {
24                  console.log('Error: ' + err.stack);
25                  dialog.showErrorBox("Error:", 'Wrong credentials');
26              }
27              else {
28                  //Guardamos el token
29                  window.token = token;
30                  window.user = user;
31                  if (user.type == 'owner') {
32                      navigateTo('owner');
33                  } else if (user.type == 'employee') {
34                      navigateTo('choose');
35                  } else {
36                      navigateTo('home');
37                  }
38              }
39          });
```

Figure 16: Function renderer.js made by the author

On e.g. figure 16 have the functionalities to show and hide the login view, also we have the function login, when it is called from the view and how take the data from the view

```
renderer.js M ×
cliente > app > www > js > renderer.js > editOrder
577  pages.ticket = {
578      show: function (data) {
579          console.log('ticket.show()');
580
581          printTicket(token, token, (err, productList) => {
582              if (err) {
583                  console.log('Error: ' + err.stack);
584                  dialog.showErrorBox("Error:", 'Error showing products');
585              }
586              else {
587
588                  let totalPrice = 0;
589                  let htmlTicket = ``;
590                  let htmlPaymentMethod = ``;
591                  productList.forEach((product) => {
592                      let finalPrice = product.qty * product.price;
593                      totalPrice += finalPrice;
594 >                    htmlTicket += `···
609                      `;
610                      htmlPaymentMethod = `
611                      <a onclick="pages.ticket.paymentMethod('${product.idOrder}','cash')" class="waves-effect
612                      <a onclick="pages.ticket.paymentMethod('${product.idOrder}','card')" class="waves-effect
613                      `;
614                  });
615                  let htmlTotalPrice = `<h5> ${totalPrice} \u20AC</h5>`;
616                  document.getElementById('ticket').innerHTML = htmlTicket;
617                  document.getElementById('finalPrice').innerHTML = htmlTotalPrice;
618                  document.getElementById('paymentMethods').innerHTML = htmlPaymentMethod;
619              }
620          });
```

Figure 17: Function 2 renderer.js made by the author

On e.g. figure 17 It showed a function in how we introduce dynamic data, for print ticket first we need to know the product in the order, after execute the function to get this info we contact the html variable and put the information we get for the function, finally we look for the element on the original view and insert our html variable with the variable we created. Also in the html code that we insert it is possible to see how we call the function from the view, I must follow the next notation, “pages.name_of_view.method”.

4.1.2. Use RESTful service

The last part of the client is to check how our system uses the RESTful service implemented on the server side. This functionality is developed in the model_rest.js file. Here I use the axios libraries to make the requests to the microservices depending on which operation I use the URL will change to follow our RESTful API, also as we explain before we have 2 different microservices, then the URL change depend on the microservice we call from here.

```
JS model_rest.js ×
cliente > app > www > js > JS model_rest.js > ...
1   const axios = require('axios');
2   const urlBaseUser = 'http://localhost:8090/e-order';
3   const urlBaseProduct = 'http://localhost:8080/e-order';
4
5
6   function login(name, password, cb) {
7       axios.post(urlBaseUser + '/sessions',
8           { name: name, password: password })
9           .then(res => {
10              cb(null, res.data.token, res.data.user)
11          })
12          .catch(err => {
13              cb(err);
14          });
15  }
```

Figure 18: Model_rest Client made by author

4.2. Implementation Server Side

In this section I explain how to implement the client and restaurant microservice, the microservices are make by file with the API RESTful listening in a port, the operations with the database and the image for docker to deploy the microservice, then the structure for microservices will be similar to:

microservice/
| - dockerfile
| - model_microservice.js
| - microservice.js

4.2.1. Microservice Client

The main purpose of client microservice it is to handle the information of all types of users (employee, owner and customers), also these microservices handle the orders associated with the customers.

4.2.1.1. API RESTful

In this file I implement all the function associated with the client microservice, that I mentioned before e.g Table 6, then for access to this service

```
34  app.use(function (req, res, next) {
35      console.log('authorize ' + req.method + ' ' + req.originalUrl);
36
37
38      if ((req.path == '/e-order/sessions' && req.method == 'POST') ||
39          (req.path == '/e-order/users' && req.method == 'POST')) {
40          next();
41      } else if (!req.query.token) {
42          res.status(401).send('Token not found');
43      }
44      else next();
45  });
46
47  //Funcion de Login Ok
48  app.post('/e-order/sessions', function (req, res) {
49      console.log('login ' + JSON.stringify(req.body));
50      if (!req.body.name || !req.body.password) {
51          res.status(400).send('Parameters missing');
52      }
53      else {
54          model.login(req.body.name, req.body.password, (err, token, user) => {
55              if (err) {
56                  res.status(400).send(err);
57              } else {
58                  res.send({ token: token, user: user });
59              }
60          });
61      }
62  });
```

Figure 19: API RESTful made by author

In e.g figure 19 the first function it is for authentication for every operation it is required they have the token that they take when the user does the login, only the function to register users and login don't need the token to do the operation, also the last function is an example as how it is implemented the API, we put a post method, and it is called when the URL it is '/e-order/session' also it is parsed in body the info from the users, after take the information, the function login it is called, finally if the parameters are correct then we receive the user and his token, else error is sending

4.2.1.2. Database operations (CRUD implementation)

In this file, using the MongoDB operation[16], I implement the API for e.g table 4, there is an example of CRUD operation with the database

In this operation is adding a new type

```
//Ok
function addType(token, type, cb) {
    MongoClient.connect(url, function (err, client) {
        if (err) cb(err);
        else {
            console.log('connected');
            // create new callback for closing connection
            _cb = function (err, res) {
                client.close();
                cb(err, res);
            }
            let db = client.db('e-order');
            let product = db.collection('product');

            product.findOne({ type: type.type },
                (err, _product) => {
                    if (err) _cb(err);
                    else if (_product) _cb(new Error('Type already exists'));
                    else {
                        type['type'] = type.type.toLowerCase();
                        type['lstPr'] = [];
                        product.insertOne(type, (err, result) => {
                            if (err) _cb(err);
                            else {
                                _cb(null, { id: result.insertedId.toHexString(), name: type.name, lstPr: type.lstPr });
                            }
                        });
                    }
                });
        }
    });
}
```

Figure 20 a: Example API create operation made by author

In this operation it is reading the information about the orders still open and each order is saved in _orders, which variable it is returned in the callback

```
375  function openOrders(token, cb) {
376      MongoClient.connect(url, function (err, client) {
377          if (err) cb(err)
378
379          console.log('connected.');
380          function _cb(err, result) {
381              client.close();
382              cb(err, result);
383          }
384
385          var db = client.db('e-order');
386          var colCustomer = db.collection('customer');
387
388          var userOrder = [];
389
390          colCustomer.find({ paid: false }).toArray((err, _orders) => {
391              if (err) _cb(err);
392              else {
393                  _orders.forEach((_order) => {
394                      userOrder.push({
395                          refClient: _order.refClient,
396                          num_table: _order.num_table,
397                          methodPay: _order.methodPay
398                      });
399                  });
400                  _cb(null, userOrder)
401              }
402          });
403      });
404  }
```

Figure 20 b: Example API read operation made by author

In this operation is updating some product in the order that receive

```
260  function editOrder(token, orderID, content, cb) {
261
262      MongoClient.connect(url, function (err, client) {
263          if (err) cb(err);
264          else {
265              console.log('connected.');
266              function _cb(err, result) {
267                  client.close();
268                  cb(err, result);
269              }
270
271              var db = client.db('e-order');
272              var col = db.collection('customer');
273
274              if (!content.methodPay) {
275                  col.findOne({ _id: new mongodb.ObjectId(orderID) }, (err, _order) => {
276                      if (err) _cb(err);
277                      else {
278                          console.log(_order);
279                          col.updateOne({ _id: new mongodb.ObjectId(orderID), "productOrder.productId": content.productId }, {
280                              $set: {
281                                  "productOrder.$.productId": content.productId,
282                                  "productOrder.$.qty": content.qty,
283                                  "productOrder.$.notes": content.notes
284                              }
285                          }, (err, result) => {
286                              if (err) _cb(err);
287                              else {
288                                  _cb(null, ('Data updated'))
289                              }
290                          });
291                      }
292                  });
293 >            } else {···
302              }
303          }
```

Figure 20 c: Example API update operation made by author

In this operations is deleting the order

```
308  //Ok
309  function cancelOrder(token, orderID, cb) {
310      MongoClient.connect(url, function (err, client) {
311          if (err) cb(err);
312          else {
313              console.log('connected.');
314              function _cb(err, result) {
315                  client.close();
316                  cb(err, result);
317              }
318
319              var db = client.db('e-order');
320              var colCustomer = db.collection('customer');
321
322              colCustomer.findOne({ _id: new mongodb.ObjectId(orderID) }, (err, _order) => {
323                  if (err) _cb(err);
324                  else {
325                      colCustomer.findOneAndUpdate({ _id: new mongodb.ObjectId(_order.refClient) }, {
326                          $pull: { orders: orderID }
327                      }, (err, result) => {
328                          if (err) _cb(err);
329                          else {
330                              colCustomer.deleteOne({ _id: new mongodb.ObjectId(orderID) }, (err, _result) => {
331                                  if (err) _cb(err);
332                                  else {
333                                      _cb(null, 'Order deleted')
334                                  }
335                              });
336                          }
337                      });
338                  }
339              });
340          }
341
342      });
343  }
```

Figure 20 d: Example API delete operation made by author

It is important to know each function It is required to open the communication with the database (MongoClient.connect()), and we finish the operation we call the function _cb that return the cb with the result and close the connection with the database (client.close())

4.2.1.3. Dockerfile

The file Dockerfile contains the information on how to create the new image for our microservice. Then on the e.g. figure 21, The image start from the node image available on Docker Hub, after It added some label for information, also our microservice accept two parameters for configuration, the first one it is the port where our microservice will be listening and the second it is the URL for connecting with the MongoDB database. Finally, it is added all the files to the microservice works on the folder /app and create a new folder inside, /images for store the images, also it is installed on all the dependencies and it is defined the entry point with the parameters mentioned before.

```
users > dockerfile
1   FROM node
2
3   LABEL description="Users microservice"
4   LABEL version="1.0"
5   LABEL maintainer="danymexicano5@gmail.com"
6
7   ENV PORT=8090
8   ENV URL_MONGODB="mongodb://127.0.0.1:27017"
9
10  ADD users.js model_users.js package.json /app/
11  RUN mkdir -p /app/images
12  RUN npm --prefix /app install
13  ENTRYPOINT node /app $PORT $URL_MONGODB
```

Figure 21: Dockerfile Users made by author

4.2.2. Microservice Restaurant

The main purpose of restaurant microservice is to handle the information of the type and products.

4.2.2.1. API RESTful

In this file is similar to the other microservice, it implements all the functions of this microservice from e.g table6, to be called from the client.

```
17  const fileStorageEngineProduct = multer.diskStorage({
18      destination: (req, file, cb) => {
19          cb(null, "./app/images");
20      },
21      filename: (req, file, cb) => {
22          cb(null, Date.now() + '--' + file.originalname);
23      }
24  });
25
26  const uploadPr = multer({ storage: fileStorageEngineProduct });
```

Figure 22: API RESTful restaurant made by author

In e.g figure 22 this function is how to introduce images for the products, first it is required the multer[17] library to store the images, with this library it is possible to choose where it is possible to store the data and the filename, for the microservice the images will be safe in a folder /app/images(the folder /app will be created on the dockerfile e.g figure 24) and the name for the files will be the current date with the name from the file, then we define a const for upload the picture. Then our function to store images is ready and we could access these images to put in our client.

4.2.2.2. Database Operations

In this file it is similar to the other microservice, I create a function for upload the database, but in this microservices we have other difference, If we remember in the e.g. figure 8 we have a dependency from restaurant microservice to customer microservice, that's because we need some information from the orders at the moment to print the ticket and update the sales. Then we need to create specific calls to the other microservice to have access to this data.

```
product > JS model_product.js > ...
16  function getSession(token, cb) {
17      console.log(urlUsers + '/sessions' + {
18          params: { token: token }
19      });
20      axios.get(urlUsers + '/sessions',
21          {
22              params: { token: token }
23          })
24          .then(res => {
25              cb(null, res.data)
26          })
27          .catch(err => {
28              cb(err);
29          });
30  }
31
32  function getCollection(token, id, cb) {
33      console.log(urlUsers + '/info/' + id);
34      axios.get(urlUsers + '/info/' + id,
35          {
36              params: { token: token }
37          })
38          .then(res => {
39              cb(null, res.data)
40          })
41          .catch(err => {
42              cb(err);
43          });
44  }
```

Figure 23: Calls for users microservice made by author

Then in the e.g figure 21, we have two different calls to the customers microservice, getSession to check the token from our user, and the second function getCollection with this function we can get a specific collection from the customer microservices, with this function configured our microservices restaurant it is ready to request for collection to the customer microservice

4.2.2.3. Dockerfile

In this Dockerfile it is defined as the information to create the new image for restaurant microservice, this image is very similar to the other image, but in this file we have a few differences. The difference it is on the configuration parameters, first it is this service will be listened on the 8080 port and here we have a new parameter (url_users), this parameter it is to connect the restaurant microservice to the customers microservices, if we check the url it is calling the port :8090, which is where it is the customers microservices listening for request.

```
product > dockerfile
1   FROM node
2
3   LABEL description="Product microservice"
4   LABEL version="1.0"
5   LABEL maintainer="danymexicano5@gmail.com"
6
7   ENV PORT=8080
8   ENV URL_USERS="http://host.docker.internal:8090/e-order"
9   ENV URL_MONGODB="mongodb://127.0.0.1:27017"
10
11  ADD product.js model_product.js package.json /app/
12  RUN mkdir -p /app/images
13  RUN npm --prefix /app install
14  ENTRYPOINT node /app $PORT $URL_USERS $URL_MONGODB
```

Figure 24: Dockerfile for restaurant image made by author

4.2.3. Deploy App

Now with all the systems developed we need to containerize our microservices, to deploy our system, for this I will use docker-compose.yml to deploy our microservices in different containers on the same host.

```yaml
2   services:
3     users:
4       build:
5         ./users
6       environment:
7         PORT:
8           '8090'
9         URL_MONGODB: 'mongodb://mongo1:27017'
10      ports:
11        - '8090:8090'
12      networks:
13        - lan
14      depends_on:
15        - mongo1
16      volumes:
17        - images1:/app/images
18        - ./cliente/app/www/images/users:/app/images
19
20    mongo1:
21      image: mongo
22      networks:
23        - lan
24      volumes:
25        - vol1:/data/db
26      ports:
27        - '27090:27017'
```

Figure 25: Docker-compose users and mongo1 containers mae by author

In the e.g figure 25 this part from the docker-compose.yml is how it is deployed the containers for users microservice and MongoDB, then for deploying a container docker-compose requires an image, for our microservice customer this image is in the folder ./users where we have our Dockerfile defining this image (e.g table 21), in the environment it is defined the parameters for the Dockerfile to this container is listening on the port 8090 and the connection with the MongoDB database, in port it is configure that the container is listening in the 8090 port, then this container it is attached to the LAN network and create the dependency with mongo1 for at the moment to deploy the container, mongo 1 should be created first, finally we have the volume images1, this volume it works to store the image from this service, after it is created a temporary folder with the images in the client side. In other hand the other service is mongo 1 this is the MongoDB database for users microservices, for that it is downloaded the image mongo from Docker Hub, attached to the LAN network, also it is created a volume to store all the data and the service will be

listening in the port 27090, (that's not require, but it is good to have easily access to the database and check everything run properly)

```
product:
  build:
    ./product
  environment:
    PORT:
      '8080'
    URL_USERS: 'http://host.docker.internal:8090/e-order'
    URL_MONGODB: 'mongodb://mongo2:27017'
  ports:
    - '8080:8080'
  networks:
    - lan
  depends_on:
    - users
    - mongo2
  volumes:
    - images2:/app/images
    - ./cliente/app/www/images/product:/app/images


mongo2:
  image: mongo
  networks:
    - lan
  volumes:
    - vol2:/data/db
  ports:
    - '27080:27017'
```

Figure 26: Docker-compose customer and mongo2 containers made by author

This part from the document is similar to the e.g figure 25. Here it is defined as the microservice restaurant and mongo 2, the difference is the service restaurant is listening in the port :8080 also it is defined the connection with the other microservice for that we add the users container into the depends on from this container. In addition, we have the mongo 2, this is the database for the restaurant microservices, it is the same configuration as mongo 1, the only difference it is this database is listening in the port :27080

```
58 volumes:
59   vol1:
60   vol2:
61   images1:
62   users:
63     external: true
64   images2:
65   product:
66     external: true
67 networks:
68   lan:
```

Figure 27: Docker-Compose Volumes and networks mae by author

This is the end for the docker-compose.yml, in the e.g figure 27 it is defined the volumes it is used for the containers, also the volume users and products are declared as external because this is stored on the client side. Finally it is defined the network lan

With all the docke-compose.yml defined for deploy the system, I execute the command docker-compose up

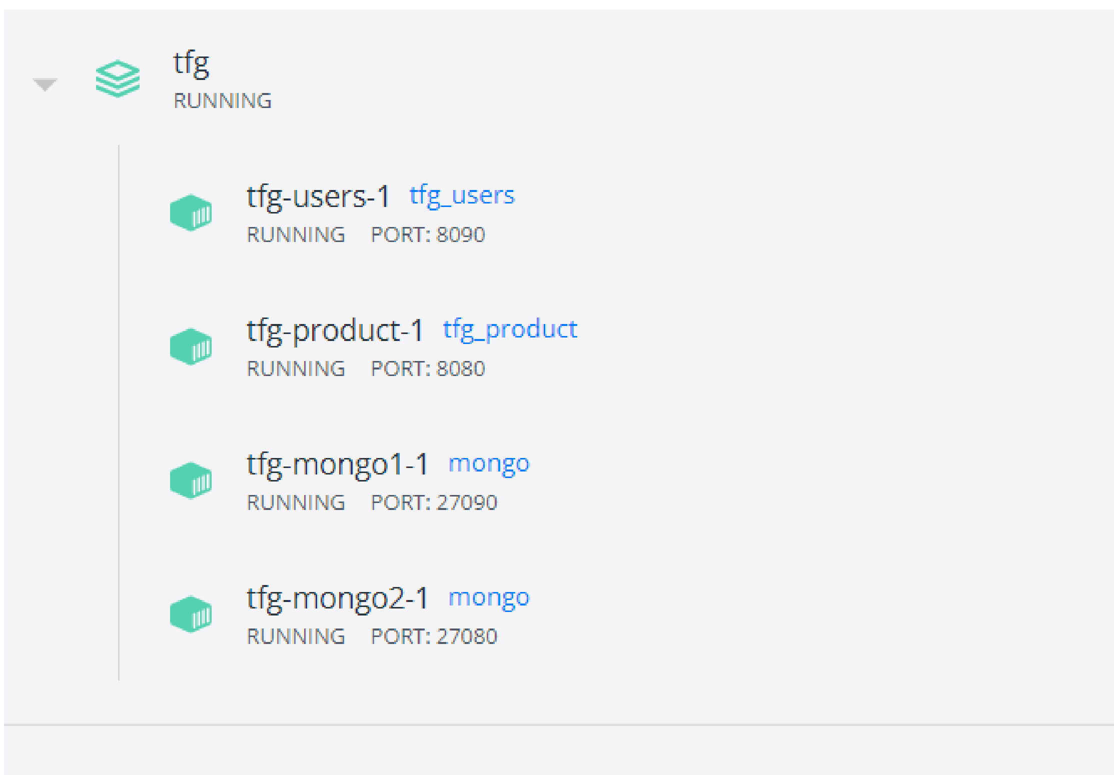


Figure 28: Container Deployed made by author

5. Results

After finishing the development of this system, I got a frontend because it is good how to take the data from the frontend and parse this information for sending to the database through to the microservices. Also, I must mention the dynamic parts in the frontend, for example the menu in the restaurant.

Also, the development of the microservices, how they are working to receive information from the client and how they can communicate to exchange information and how they got containerized with docker. On the other hand, the CRUD operation is implemented to handle the data on the database.

Finally, it is important to remark how it is easy to deploy the system due to the docker images and docker-compose, which we deploy the app with the specification on these files

6. Providing the testing of the system

Now with the system developed It is important to check the system works properly, then first it is important to know how the system could be deployed, for doing that it will be required to deploy the container with the microservices, the for doing that we must go where we have the docker-compose and run the command "**docker-compose up",** this command deploy our microservices in this container. After being deployed if we check the container they are ready and listening on the port that we set up e.g. figure 29

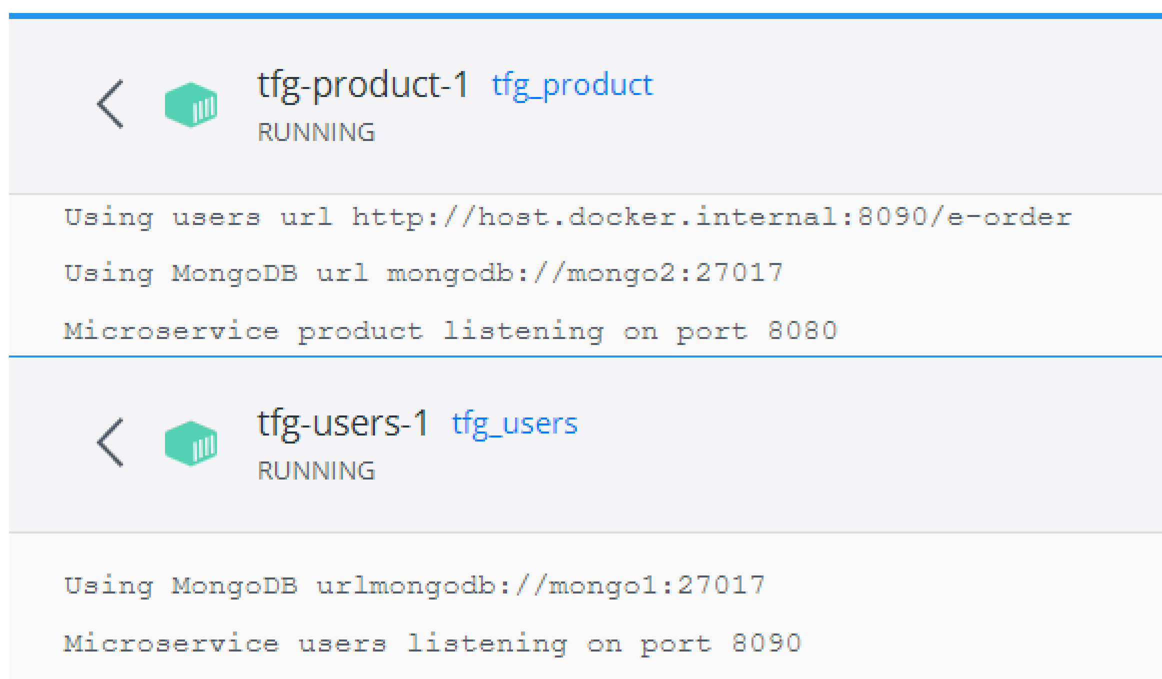


Figure 29: Running containers

After verifying that the containers are running correctly. Now it is possible to run and test our client side, then for doing that we go to the client folder and run the command “**npx electron .**”. And now our client will be open and ready to use the system.

Now it is possible to test the system, the the test will be provided as following:

1) Login
2) Register
3) Add Product and Type
4) Create, edit, delete and Update order
5) Show order and update Sale

In the e.g figure 30 first we start with registering the first user for the system, it is important to know the first user in being registered will be tagged as the owner, then we check the logs in the container how the function is call and execute the function addUser, after being register it is tried the login function which works properly. Finally, we tried the function to add an employee, and in the logs of the container it is shown how the function is called and returns the information about our new employee.

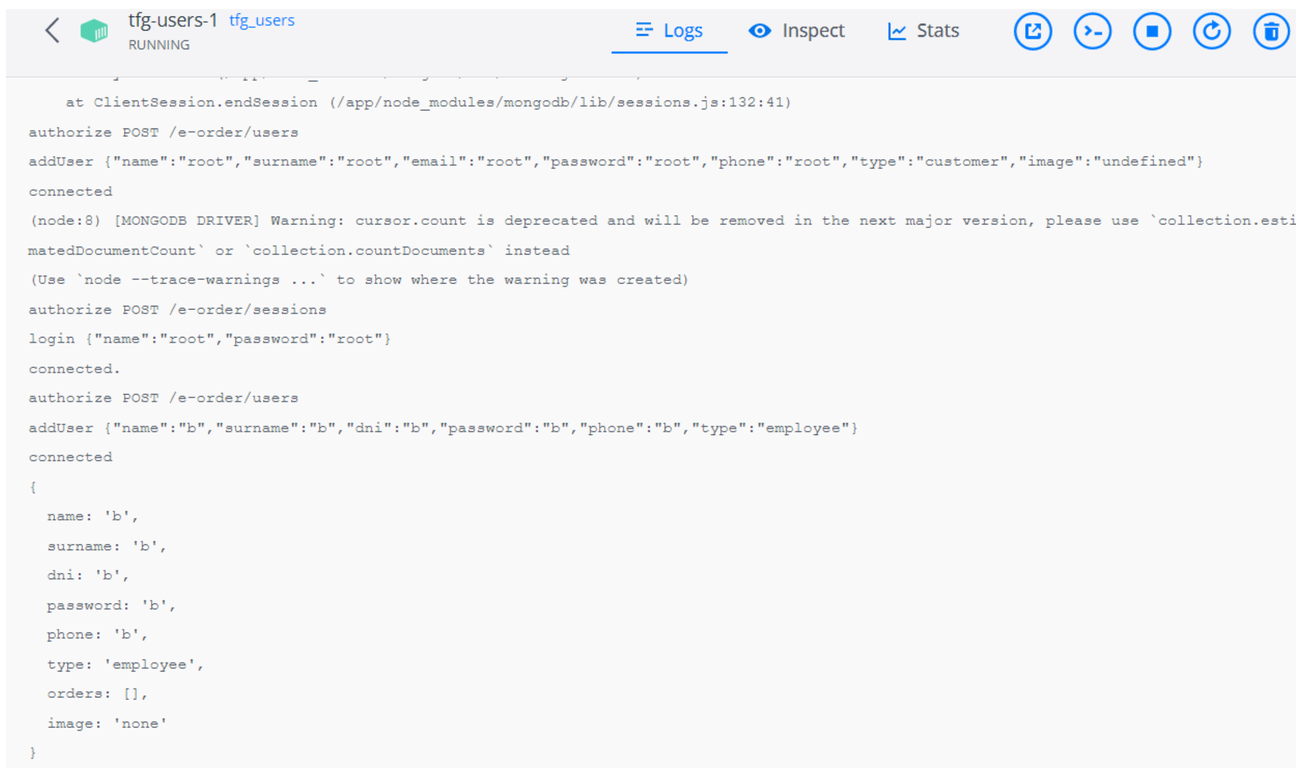


Figure 30: Microservice Users, register&&login made by the author

In the e.g figure 31 it is showed the logs of the container when we add a new type and product in the menu, if we check the logs the function it is going properly and we get the information about the new type and product, also there are some times it is called the function list menu, which is required from our client side

tfg-product-1 tfg_product
RUNNING

Logs Inspect Stats

```
List Menu
connected.
authorize GET /e-order/order?token=628bbd0ade581cd02d0866a9
List Menu
connected.
authorize GET /e-order/order?token=628bbd0ade581cd02d0866a9
List Menu
connected.
authorize POST /e-order/product/type?token=628bbd0ade581cd02d0866a9
add type{"type":"drinks"}
connected
authorize GET /e-order/order?token=628bbd0ade581cd02d0866a9
List Menu
connected.
authorize POST /e-order/product?token=628bbd0ade581cd02d0866a9
add product{"name":"Water","price":"1","typePr":"drinks","description":"Stay hidrated"}
[Object: null prototype] {
  name: 'Water',
  price: '1',
  typePr: 'drinks',
  description: 'Stay hidrated',
  image: 'images/product/1653325777838--pexels-alex-azabache-3766180.jpg'
}
```

Figure 31 Product Microservice add type && product made by the author

In the e.g figure 32 it is possible to view the logs for how we create, edit and cancel the order, also we can see which product it is added to the order and some extra data, the same for when we edit the order, but when the order is canceled only the ID from the order is require. Moreover, there are some other logs, check token and getCollection. If we remember this function is called from the product microservice, that's because the client needs to show the product from the order.

tfg-users-1 tfg_users
RUNNING
Logs Inspect Stats

```
authorize POST /e-order/order?token=628bc1aade581cd02d0866ab
create order{"productId":"628bbfd2d834a71eb5b0da1d","qty":3,"notes":""}
connected.
authorize GET /e-order/sessions?token=628bc1aade581cd02d0866ab
checkToken 628bc1aade581cd02d0866ab
connected.
authorize GET /e-order/info/628bc1aade581cd02d0866ab?token=628bc1aade581cd02d0866ab
Get collection: 628bc1aade581cd02d0866ab
connected.
authorize GET /e-order/info/628bc1b5de581cd02d0866ac?token=628bc1aade581cd02d0866ab
Get collection: 628bc1b5de581cd02d0866ac
connected.
authorize DELETE /e-order/order/628bc1b5de581cd02d0866ac?token=628bc1aade581cd02d0866ab
delete order 628bc1b5de581cd02d0866ac
connected.
authorize DELETE /e-order/order/628bc1b5de581cd02d0866ac?token=628bc1aade581cd02d0866ab
delete order 628bc1b5de581cd02d0866ac
connected.
```

Figure 32: Create, edit and cancel order made by the author

In the e.g. figure 33 we can see our last tests, first in users microservice we order the food and want to pay, then for the system we change the method Pay from the order to the method taken by the client, also we have the other function open orders that is because the client need to show this order to the employee in the dining view. Finally, in the microservice product when the order is finished, we need to update the sales from this product.

tfg-users-1 tfg_users
RUNNING
Logs Inspect Stats

```
authorize PUT /e-order/order/628bc531de581cd02d0866ad?token=628bc1aade581cd02d0866ab
edit order {"methodPay":"card"}
connected.
authorize POST /e-order/sessions
authorize GET /e-order/order/open?token=628bbd52de581cd02d0866aa
open orders
connected.
```

tfg-product-1 tfg_product
RUNNING
Logs Inspect Stats

```
authorize PUT /e-order/order/sales/628bc531de581cd02d0866ad?token=628bbd52de581cd02d0866aa
update sales
628bc531de581cd02d0866ad
http://host.docker.internal:8090/e-order/sessions[object Object]
connected.
http://host.docker.internal:8090/e-order/info/628bc531de581cd02d0866ad
```

Figure 33: Order, Show orders and update orders made by the author

We can see that the system works correctly, both the calls from the client and the management of its microservices, although I think it would be necessary to test it with many users to be able to confirm this data.

7. Conclusions:

Once the development of the application is finish, it is time to make an analysis of the final system and how it could be improved. First I think it would be important to consider our objective, the phases of development and the final result.

The analysis of the app it was very useful, because you have a first idea to how will be the system, only just need to follow the indications at the time to develop the system, therefore it was good to have a deep analysis of how will be the frontend and the backend of the system. Sometimes in the development of a system this is the most significant part to not waste too much time in redesigning some parts of the system.

About the system it was good to practise how it works with microservices, how to make the CRUD operation for the database and connect those operations with the API RESTful for having the microservice ready to the requests.

Another essential part in the system was to containerize the microservices with docker, due to this it was easier to deploy the app.

Finally, it is indispensable to say the system could be improved, for example the GUI is not very friendly to the users. For example the owner and employee views are very simple and with almost no components from materialize.css, also there are no colours, all of this is worst experience for the users.

8. Perspective

Looking at the future although the system it is developed, this could be improved adding some new specs. For example, I thought adding a new microservice to handle the booking for the restaurant would be great because this adds value to the product, and it is helpful to introduce this into the market.

9. References

[0] OECD. (2020, March 04). *OECD (2020), OECD Tourism Trends and Policies 2020, OECD Publishing, Paris*. OECD Tourism Trends and Policies. Retrieved May 20, 2022, from https://www.oecd-ilibrary.org/sites/8ed5145b-en/index.html?itemId=/content/component/8ed5145b-en#chapter-d1e90998

[1] UBER-EATS - https://www.ubereats.com/

[2] INSTACART - https://www.instacart.com/

[3] STACKSHARE - Uber Eats - https://stackshare.io/uber-technologies/uber

[4] STACKSHARE - Instacart - https://stackshare.io/instacart/instacart

[5] BY FRENCH PRESS AGENCY - AFP. (2021, October 31). Spain's delivery riders law reshuffles deck for take-away market. *Daily Sabah*. https://www.dailysabah.com/business/economy/spains-delivery-riders-law-reshuffles-deck-for-take-away-market

[6] ELECTRON - https://www.electronjs.org/

[7] Springer, C. (2017, 09 06). *Microservices: Yesterday, Today, and Tomorrow*. Springer Link. Retrieved 05 21, 2022, from https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12#citeas

[8] NODE.JS - https://nodejs.org/en/about/

[9] EXPRESS - http://expressjs.com/

[10] AXIOS - https://github.com/axios/axios

[11] MATERIALIZE.CSS - https://materializecss.com/

[12] MONGODB - https://www.mongodb.com/

[13] DOCKER - https://www.docker.com/

[14]Nathan, S., Ghosh, R., Mukherjee, T., & Narayanan, K. (2017, 05 11). *CoMICon: A Co-Operative Management System for Docker Container Images*. IEEE International Conference on Cloud Engineering (IC2E). Retrieved 05 22, 2022, from https://ieeexplore.ieee.org/abstract/document/7923794

[15] DOCKER-HUB - https://hub.docker.com/

[16] MONGODB OPERATION - https://www.mongodb.com/docs/manual/introduction/

[17] MULTER - https://www.npmjs.com/package/multer

Appendix 1

Url link to the project: https://github.com/DemeX404/TFG