Summary

Introduction

This report aims to describe the structure and operation of the web application 'RESTaurant', which was developed for the 'Web Applications and Technologies' course in the academic year 2022-2023.

The application was developed as a SPA with a RESTful API and aims to manage and organise the tables and their related orders within a restaurant.

Technologies employed

The application was written entirely in TypeScript and versioned via GitHub.

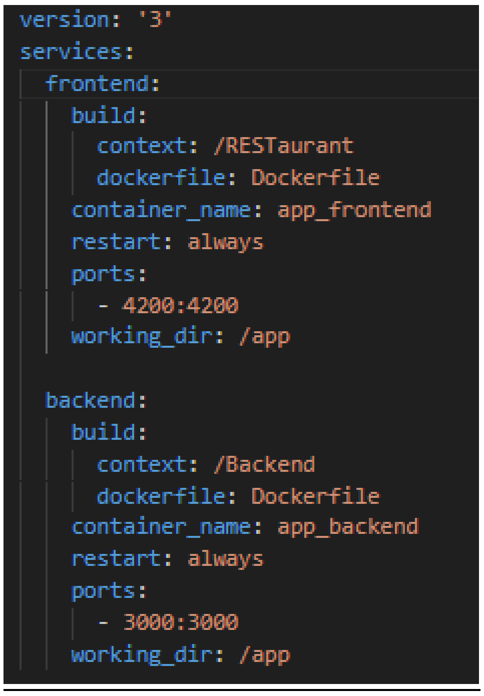
It makes use of the Node.js framework for server-side development, the Express.js framework for route definition, and the Angular framework for client-side development.
It also relies on a non-relational database, MongoDB.

In order to be able to keep all these dependency-rich components together, it was decided to create two Docker containers, one for the backend and one for the frontend of the application, managed simultaneously via Docker Compose.

Application structure and operation

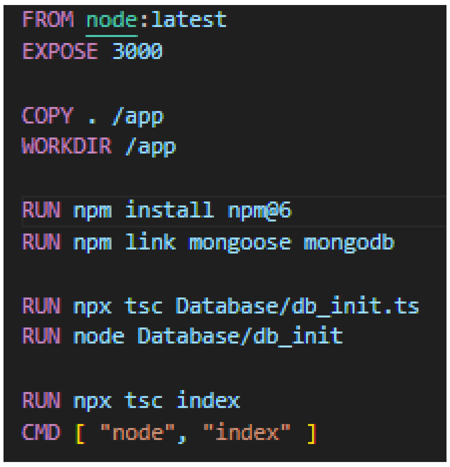
Docker Compose

Docker Compose allows the management of multiple Docker containers by creating a single YAML file.
Down below is reported the structure of our Docker Compose file:



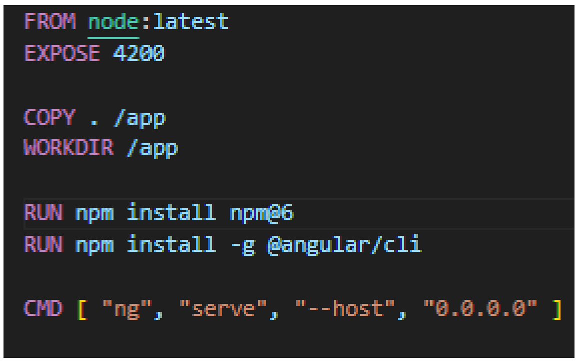
```yaml
version: '3'
services:
  frontend:
    build:
      context: /RESTaurant
      dockerfile: Dockerfile
    container_name: app_frontend
    restart: always
    ports:
      - 4200:4200
    working_dir: /app

  backend:
    build:
      context: /Backend
      dockerfile: Dockerfile
    container_name: app_backend
    restart: always
    ports:
      - 3000:3000
    working_dir: /app
```

In the Docker compose file we declared two services each corresponding to a container defined via Docker File.



```
FROM node:latest
EXPOSE 3000

COPY . /app
WORKDIR /app

RUN npm install npm@6
RUN npm link mongoose mongodb

RUN npx tsc Database/db_init.ts
RUN node Database/db_init

RUN npx tsc index
CMD [ "node", "index" ]
```

Docker File contained in Backend folder



```
FROM node:latest
EXPOSE 4200

COPY . /app
WORKDIR /app

RUN npm install npm@6
RUN npm install -g @angular/cli

CMD [ "ng", "serve", "--host", "0.0.0.0" ]
```

Docker File contained in RESTaurant folder

Both containers are created and launched with a single terminal command allowing the application to be used.

Backend

The backend of our application handles three main aspects:

1. Database
2. Routing
3. Notifications

A detailed analysis of these aspects is proposed in the following sections.

Database

RESTaurant makes use of MongoDB, a non-relational document-oriented database for the persistence of important data such as, for example, the orders of a certain table. The application makes use of the Mongoose library in order to easily relate to the database.

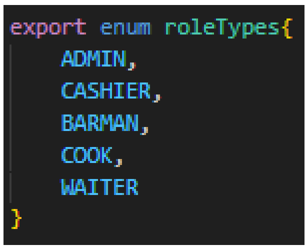
The database uses four models:

1. **User**
   This model represents the users of the application (i.e. the employees of the restaurant).

   A user is defined by:
   - **name**: the user's full name;
   - **email**: the email required to access the application;

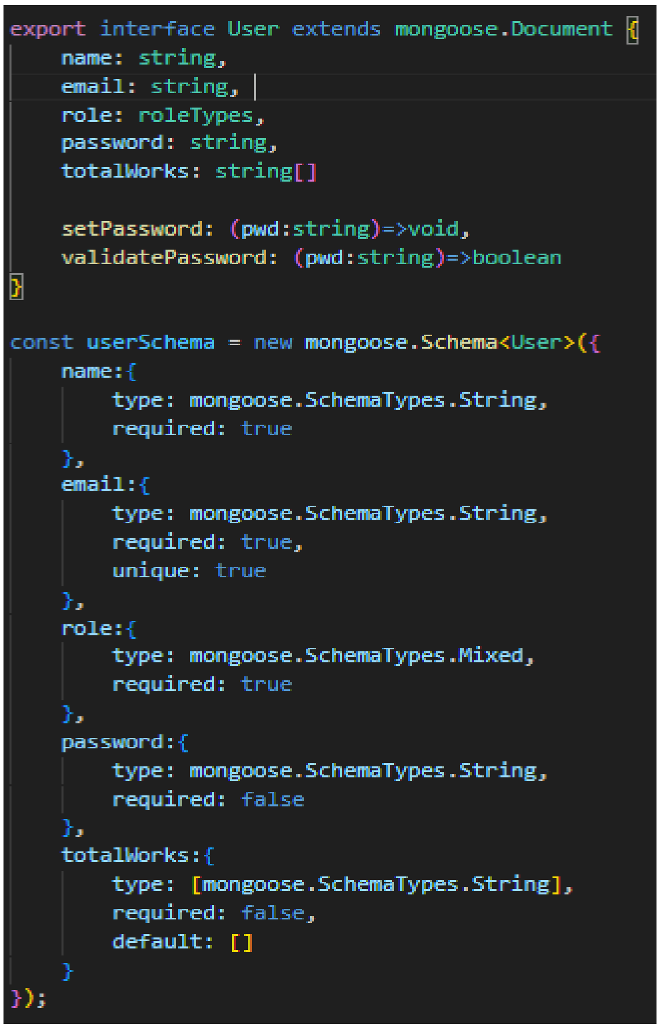
- **password**: the password required to access the application;
- **role**: the role that identifies the user's task within the restaurant (waiter, cook, barman, cashier or admin) and to which certain actions and privileges are associated;
- **totalWorks**: a list of tasks completed by the user during the work shift. The tasks saved depend on the role (i.e. the waiters' tasks will be the orders taken, the cooks' tasks will be the food prepared).

There are also two methods: setPassword and validatePassword.
The first one is used to encrypt the user's chosen password and store the obtained hash in the database instead of the unencrypted password.
The second one is used to check whether the hashed password entered during login matches the stored one.



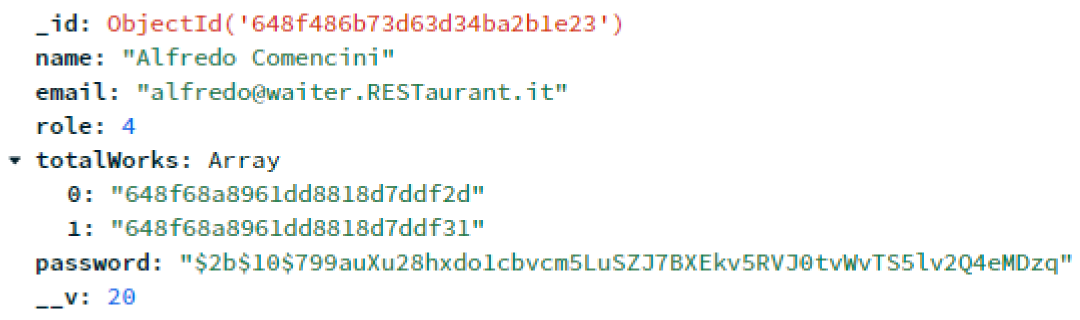
```
export enum roleTypes{
    ADMIN,
    CASHIER,
    BARMAN,
    COOK,
    WAITER
}
```

Roles assignable to a user



```
export interface User extends mongoose.Document {
    name: string,
    email: string,
    role: roleTypes,
    password: string,
    totalWorks: string[]

    setPassword: (pwd:string)=>void,
    validatePassword: (pwd:string)=>boolean
}

const userSchema = new mongoose.Schema<User>({
    name:{
        type: mongoose.SchemaTypes.String,
        required: true
    },
    email:{
        type: mongoose.SchemaTypes.String,
        required: true,
        unique: true
    },
    role:{
        type: mongoose.SchemaTypes.Mixed,
        required: true
    },
    password:{
        type: mongoose.SchemaTypes.String,
        required: false
    },
    totalWorks:{
        type: [mongoose.SchemaTypes.String],
        required: false,
        default: []
    }
});
```

User model definition



```
  _id: ObjectId('648f486b73d63d34ba2b1e23')
  name: "Alfredo Comencini"
  email: "alfredo@waiter.RESTaurant.it"
  role: 4
▾ totalWorks: Array
    0: "648f68a8961dd8818d7ddf2d"
    1: "648f68a8961dd8818d7ddf31"
  password: "$2b$10$799auXu28hxdo1cbvcm5LuSZJ7BXEkv5RVJ0tvWvTS5lv2Q4eMDzq"
  __v: 20
```

A document extracted from the "users" collection on MongoDB

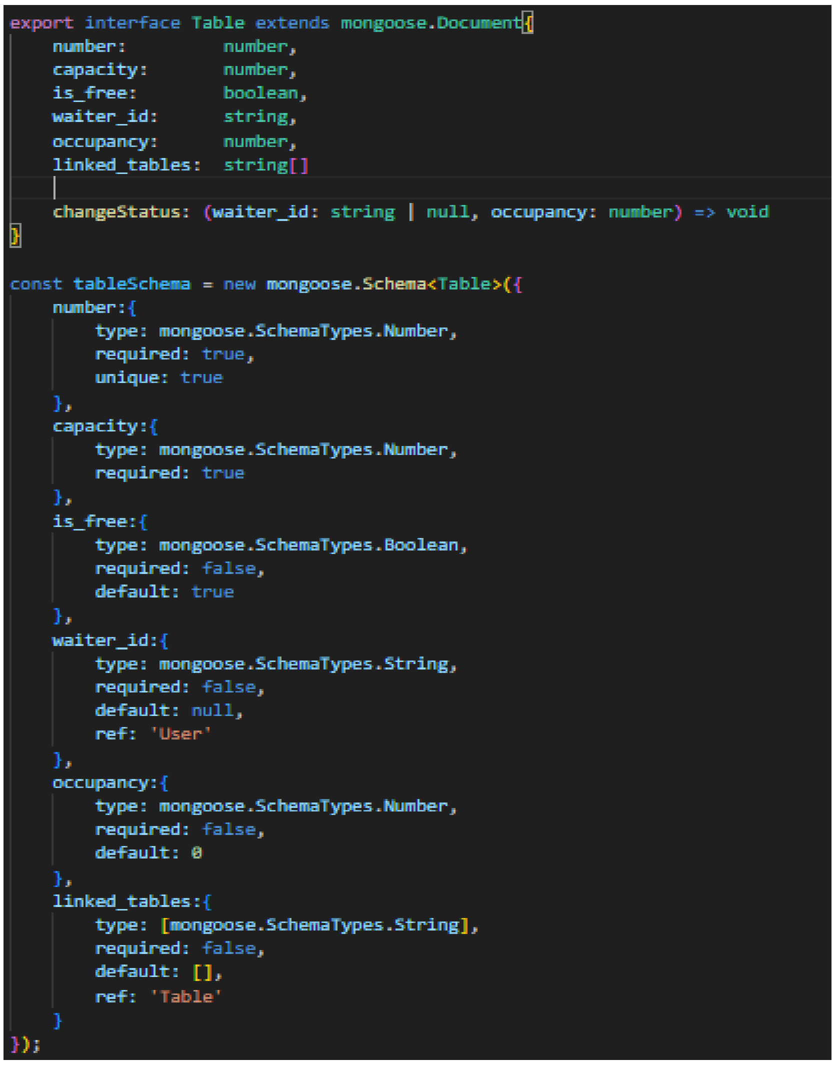
2. **Table**

This model represents a table in the restaurant.

A table is defined by:

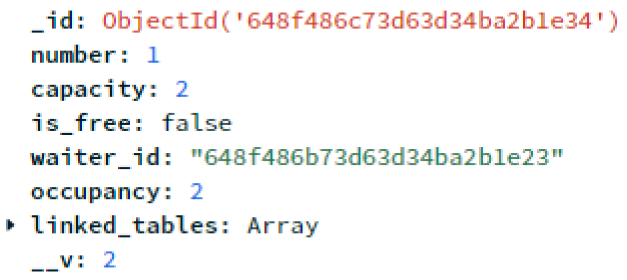
- **number**: the table identification number;
- **capacity**: the number of seats at the table;
- **is_free**: a boolean indicating whether the table is occupied by customers or not;
- **waiter_id**: the waiter who has occupied the table and who will serve it throughout the customers' stay;
- **occupancy**: the actual number of seats occupied at the table;
- **linked_tables**: a list of any tables connected to this table, which will act as the 'main table' for orders. This field has been inserted to manage the possibility of tables created by joining several tables together.

In addition, there is a changeStatus method used to free/occupy a table and store the waiter_id accordingly.



```
export interface Table extends mongoose.Document{
    number:         number,
    capacity:       number,
    is_free:        boolean,
    waiter_id:      string,
    occupancy:      number,
    linked_tables:  string[]

    changeStatus: (waiter_id: string | null, occupancy: number) => void
}

const tableSchema = new mongoose.Schema<Table>({
    number:{
        type: mongoose.SchemaTypes.Number,
        required: true,
        unique: true
    },
    capacity:{
        type: mongoose.SchemaTypes.Number,
        required: true
    },
    is_free:{
        type: mongoose.SchemaTypes.Boolean,
        required: false,
        default: true
    },
    waiter_id:{
        type: mongoose.SchemaTypes.String,
        required: false,
        default: null,
        ref: 'User'
    },
    occupancy:{
        type: mongoose.SchemaTypes.Number,
        required: false,
        default: 0
    },
    linked_tables:{
        type: [mongoose.SchemaTypes.String],
        required: false,
        default: [],
        ref: 'Table'
    }
});
```

Table model definition



```
  _id: ObjectId('648f486c73d63d34ba2b1e34')
  number: 1
  capacity: 2
  is_free: false
  waiter_id: "648f486b73d63d34ba2b1e23"
  occupancy: 2
▸ linked_tables: Array
  __v: 2
```

A document extracted from the "tables" collection on MongoDB

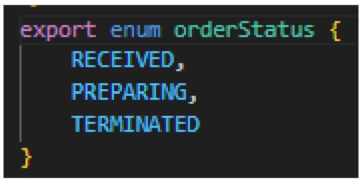
3. Order

This model represents an order taken from a certain table.

An order is defined by:

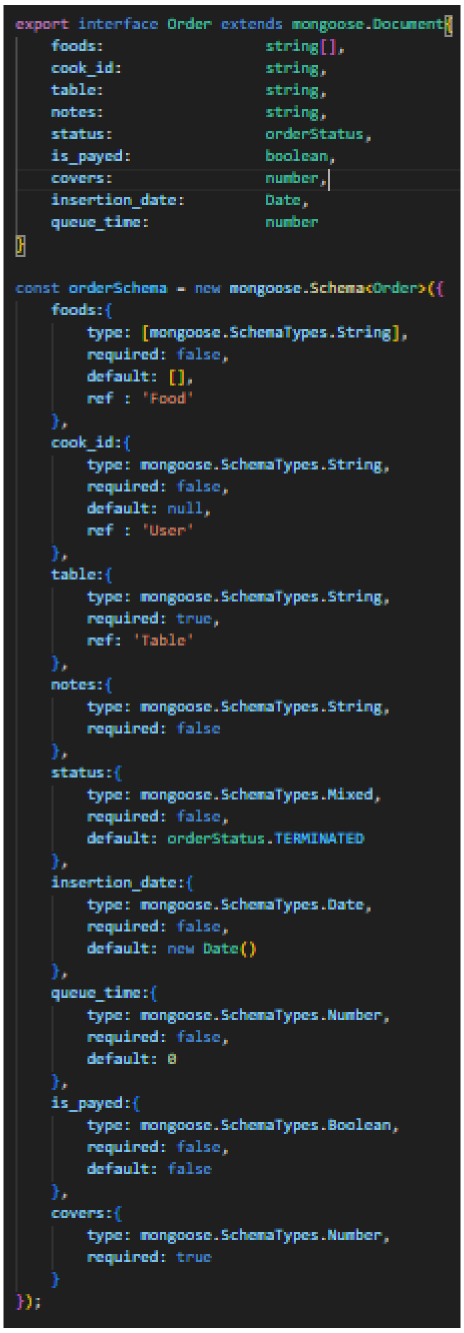
- **foods**: a list of foods/drinks to be prepared for the order;
- **cook_id**: the cook/barman who takes the order and prepares it;
- **table**: the table that placed the order. In the case of joined tables, this field will contain the "main table";
- **notes**: any additional notes to the order, such as extra ingredients or allergen;

- **status**: it represents the status of the order (received, preparing or terminated);
- **is_payed**: a boolean indicating whether a terminated order has also been paid, i.e. definitively concluded;
- **covers**: the number of covers, useful for the final receipt;
- **insertion_date**: the date of order creation;
- **queue_time**: the time needed to prepare the order.



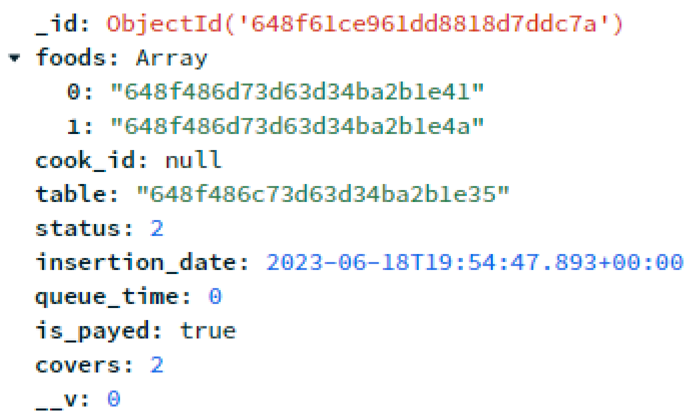
```
export enum orderStatus {
    RECEIVED,
    PREPARING,
    TERMINATED
}
```

The various states an order may be in



```
export interface Order extends mongoose.Document{
    foods:                  string[],
    cook_id:                string,
    table:                  string,
    notes:                  string,
    status:                 orderStatus,
    is_payed:               boolean,
    covers:                 number,
    insertion_date:         Date,
    queue_time:             number
}

const orderSchema = new mongoose.Schema<Order>({
    foods:{
        type: [mongoose.SchemaTypes.String],
        required: false,
        default: [],
        ref : 'Food'
    },
    cook_id:{
        type: mongoose.SchemaTypes.String,
        required: false,
        default: null,
        ref : 'User'
    },
    table:{
        type: mongoose.SchemaTypes.String,
        required: true,
        ref: 'Table'
    },
    notes:{
        type: mongoose.SchemaTypes.String,
        required: false
    },
    status:{
        type: mongoose.SchemaTypes.Mixed,
        required: false,
        default: orderStatus.TERMINATED
    },
    insertion_date:{
        type: mongoose.SchemaTypes.Date,
        required: false,
        default: new Date()
    },
    queue_time:{
        type: mongoose.SchemaTypes.Number,
        required: false,
        default: 0
    },
    is_payed:{
        type: mongoose.SchemaTypes.Boolean,
        required: false,
        default: false
    },
    covers:{
        type: mongoose.SchemaTypes.Number,
        required: true
    }
});
```

Order model definition



```
  _id: ObjectId('648f61ce961dd8818d7ddc7a')
▾ foods: Array
    0: "648f486d73d63d34ba2b1e41"
    1: "648f486d73d63d34ba2b1e4a"
  cook_id: null
  table: "648f486c73d63d34ba2b1e35"
  status: 2
  insertion_date: 2023-06-18T19:54:47.893+00:00
  queue_time: 0
  is_payed: true
  covers: 2
  __v: 0
```

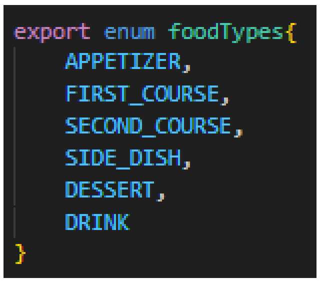
A document extracted from the "orders" collection on MongoDB

4. Food

This model represents a food/drink on the menu.

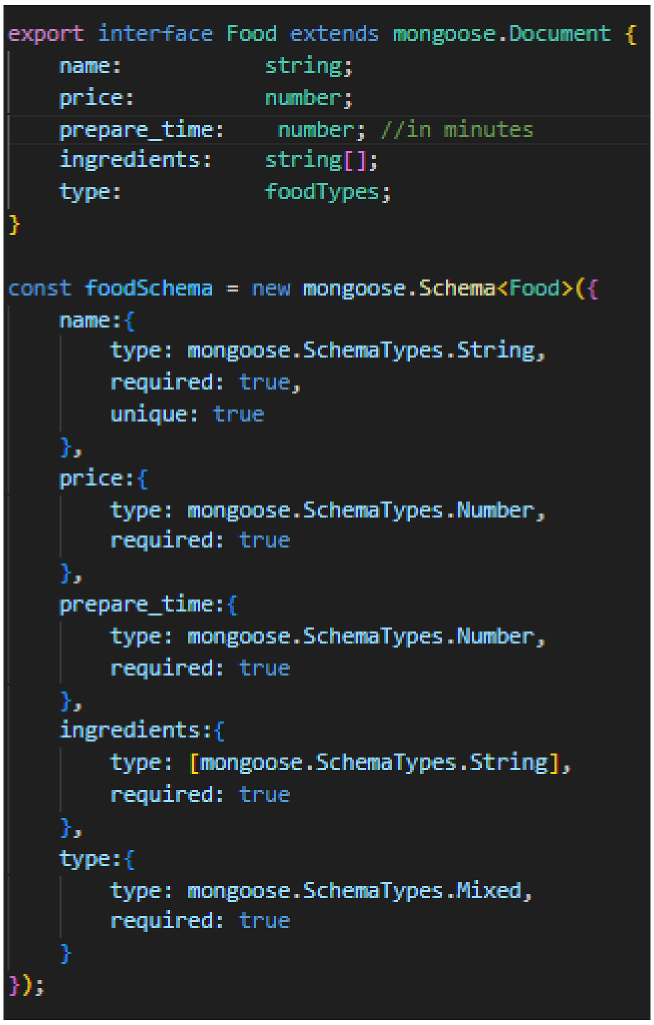
A food/drink is defined by:

- **name**: its name;
- **price**: its price;
- **prepare_time**: the time required for its preparation;
- **ingredients**: the list of its ingredients;
- **type**: its type.



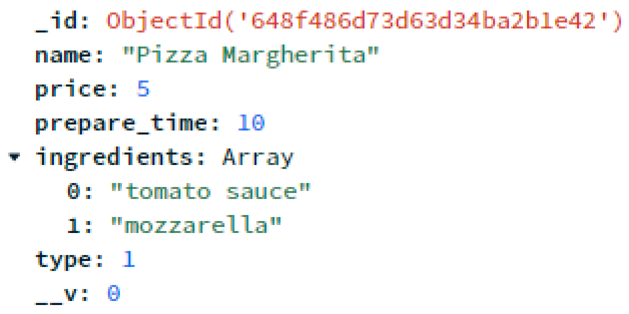
```
export enum foodTypes{
    APPETIZER,
    FIRST_COURSE,
    SECOND_COURSE,
    SIDE_DISH,
    DESSERT,
    DRINK
}
```

Types assignable to a dish/drink



```
export interface Food extends mongoose.Document {
    name:           string;
    price:          number;
    prepare_time:    number; //in minutes
    ingredients:    string[];
    type:           foodTypes;
}

const foodSchema = new mongoose.Schema<Food>({
    name:{
        type: mongoose.SchemaTypes.String,
        required: true,
        unique: true
    },
    price:{
        type: mongoose.SchemaTypes.Number,
        required: true
    },
    prepare_time:{
        type: mongoose.SchemaTypes.Number,
        required: true
    },
    ingredients:{
        type: [mongoose.SchemaTypes.String],
        required: true
    },
    type:{
        type: mongoose.SchemaTypes.Mixed,
        required: true
    }
});
```

Food model definition



```
_id: ObjectId('648f486d73d63d34ba2b1e42')
name: "Pizza Margherita"
price: 5
prepare_time: 10
▾ ingredients: Array
    0: "tomato sauce"
    1: "mozzarella"
type: 1
__v: 0
```

A document extracted from the "food" collection on MongoDB

Routing

The beating heart of the application are the routes. Through the routes users can retrieve data from the database to read and/or write them.

The server handling the requests listens on port 3000 and each request uses the HTTPS protocol (the necessary certificates have been auto-generated).

Each route makes use of an authentication middleware. The authentication middleware is needed to identify the user in the database in the case of a login, while it is needed to verify the authentication of the generated JWT and the authorisation to perform that request/action in the case of all other routes.

There are four macro-groups for the routes and each of them is associated with a database model.

The login is managed by a route in the index.ts file, which is the entry point of the server side of our application.

The four macro-groups of the routes are listed below:

- **Login**

Endpoint	Method	Authorised Users	Parameters	Description	Req contains..	Res contains..
/login	GET	Any user who has valid credentials		It checks that the credentials entered are correct. If so, it generates a valid JWT for the next 9h	Email Password	A JWT

- **User**

Endpoint	Method	Authorised Users	Parameters	Description	Req contains..	Res contains..
/users	GET	Admin, Cashier		It returns a list of all users		A list of users
/users	POST	Admin		It allows the creation of a new user	Name Email Role Password	The newly created user
/users/:user_id	PUT	Admin	user_id	It allows the updating of informations related to a certain user, identified by his id	Name Email Password Role (all optional)	The updatetd user
/users/:user_id	DELETE	Admin	user_id	It allows the deletion from the database of a certain user, identified by its id		A message “User deleted”

- **Table**

Endpoint	Method	Authorised Users	Parameters	Description	Req contains..	Res contains ..
/tables	GET	Admin, Cashier, Waiter		It returns a list of all tables		A list of tables
/tables/serving	GET	Waiter		It returns a list of tables currently served by a certain waiter, identified by his id	Id (waiter)	A list of tables
/tables/:table_id	GET	Everyone	table_id	It returns a single table, identified by its id		A table
/tables	POST	Admin		It allows the creation of a new table	Number Capacity	The newly created table
/tables	PUT	Admin, Cashier, Waiter		It allows the occupation/release of a table, identified by its id	Table_id Id (waiter) Occupancy	
/tables/link	PUT	Admin, Cashier, Waiter		It allows tables to be added to a 'main table' to create a joined table	Tables	A table full of linked tables
/tables/:table_id	PATCH	Admin	table_id	It allows the updating of informations of a certain table, identified by its id	Table_number Table_capacity (all optional)	The updated table
/tables/:table_id	DELETE	Admin	table_id	It allows the deletion from the database of a certain table, identified by its id		A message “Table deleted”

- **Order**

Endpoint	Method	Authorised Users	Parameters	Description	Req contains..	Res contains..
/orders	GET	Everyone		It returns a list of orders, different depending on the role	Role	A list of orders
/orders/all	GET	Admin, Cashier		It returns a list of all orders		A list of orders
/orders/receipt/:table_id	GET	Cashier	table_id	It allows the receipt to be calculated for a certain table, identified by its id		The receipt
/orders/totalprofit	GET	Admin, Cashier		It allows the total profit of the day to be calculated		The total profit
/orders	POST	Waiter		It allows the creation of a new order	Table Occupancy Foods	The newly created order
/orders	PUT	Cook, Barman, Cashier		It allows updating the information of a certain order, identified by its id. The updated information depends on the role	Role Id (user) Order_id	The updated order

/orders/:orderId	DELETE	Admin	orderId	It allows the deletion from the database of a certain order, identified by its id, and the deletion of orders older than two weeks		A message “Order deleted”/”Older order deleted”

- **Food**

Endpoint	Method	Authorised Users	Parameters	Description	Req contains..	Res contains..
/foods	GET	Admin, Waiter, Cashier		It returns a list of all food and drinks on the menu		A list of food and drinks
/foods	POST	Admin		It allows a new food/drink to be added to the menu	Name Price Ingredients Prepare_time Type	The newly created food/drink
/foods/:food_id	PATCH	Admin	food_id	It allows the updating of information on a certain food/drink, identified by its id	Name Price Ingredients Prepare_time Type (all optional)	The updated food/drink
/foods/:food_id	DELETE	Admin	food_id	It allows the deletion from the database of a certain food/drink, identified by its id		A message ”Food deleted”

Notifications

In line with the requirements of the project specifications, a notification system was created to signal the occurrence of certain events to certain users.

In particular, we defined the following events:

Event name	Description	Trigger
UPDATE_USERS_LIST	The user list has been modified	A user has been created/deleted/modified
UPDATE_ORDERS_LIST	The order list has been modified	An order has been created/deleted/changed status/paid
UPDATE_FOODS_LIST	The food and drink list has been modified	A food/drink has been created/deleted/modified
UPDATE_TABLES_LIST	The table list has been modified	A table has been occupied/released/modifed/deleted/created
UPDATE_TOTAL_PROFIT	The daily total profit changed	A cashier charged an order
FORCE_LOGOUT	A logged-in user is sent back to the login	A user has been deleted or its credentials changed
NEW_ORDER_RECEIVED	A new order was created and sent to the kitchen/bar	A waiter took an order
NEW_ORDER_PREPARED	An order has been prepared and is ready to be served, the waiter who took the order is notified	The cook/barman who took over the order marks it as terminated

Frontend

The frontend of the application was fully written in Angular using the MVC (Model View Controller) programming pattern, in which:

- The models are responsible for preserving useful data for the application, which can be read and updated at any time. In our case, they are represented by the database models;
- The views just take care of displaying the data taken from the models to the user, providing elements to edit the data (buttons, forms, etc.);
- Controllers are the bridge between models and views. The controllers are the ones who collect data from models and supply it to views, and they are the ones who update model data, based on user interaction with views.

In Angular, the View and Controller concepts are merged into one large block called Component.

A Component usually contains:

- An HTML template with an associated CSS file (View);
- A TypeScript file that models the behaviour of the template and its elements (Controller).

The strength of having these elements closely linked together consists in the possibility of declaring variables in the Controller, and refer them in the Views.

In addition, the Views possess special constructs such as for loops, which allow a list to be generated in a few lines, or conditional ifs, which allow the manipulation of View elements depending on certain conditions.

It is important to emphasise that the Controllers (the TypeScript files of the Components) rely on Services to retrieve data.

In fact, the Services are responsible for the generation and performance of the actual requests of the server to retrieve data. A Controller's method merely 'subscribes' to the corresponding method in the Service, which, once it has retrieved the necessary data, 'notifies' the Controller, which can then continue processing the data.

A list of the generated components and the services is presented down below:

Component name	Services used	Description
login.component	UsersService	It manages the login phase
home.component	UsersService, SocketService, OrdersService	It manages what is displayed on the home screen and notifications, all depending on the user role
food.component	FoodsService, UsersService, OrdersService, SocketService	It manages both the insertion, deletion, updating of food/beverages in the database and the addition of food/beverages to an order, which is created here
order.component	TablesService, UsersService, OrdersService, SocketService	It manages the creation, updating, and cancellation of an order, the retrieval of all orders and their division according to status, and the creation of the receipt
receipt-dialog.component		A simple dialog showing the receipt information
table.component	TablesService, UsersService, SocketService	It manages the creation, updating, deletion of a table and the retrieval of tables
table-occupancy-dialog.componenet		A simple dialog in which to enter the number of customers who will actually occupy the table
users.component	UsersService, SocketService	It manages the creation, updating, deletion of a user and the recovery of users
stats.component	TablesService, UsersService, OrdersService, FoodsService	It manages the creation and display of statistics

Lastly, it is significant to mention that Angular Material elements and guidelines were used for the styling of the views, while the chart.js library was used for the statistics graphs.

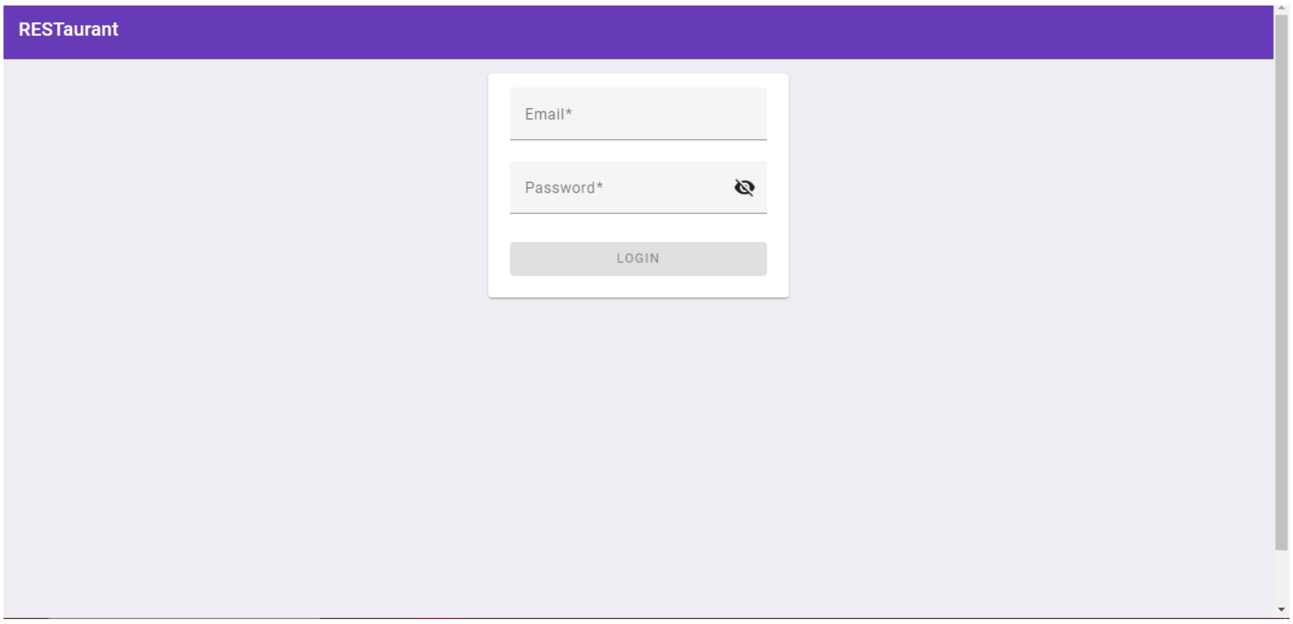
Possible upgrades

A list of the possible upgrades of the application is presented down below:

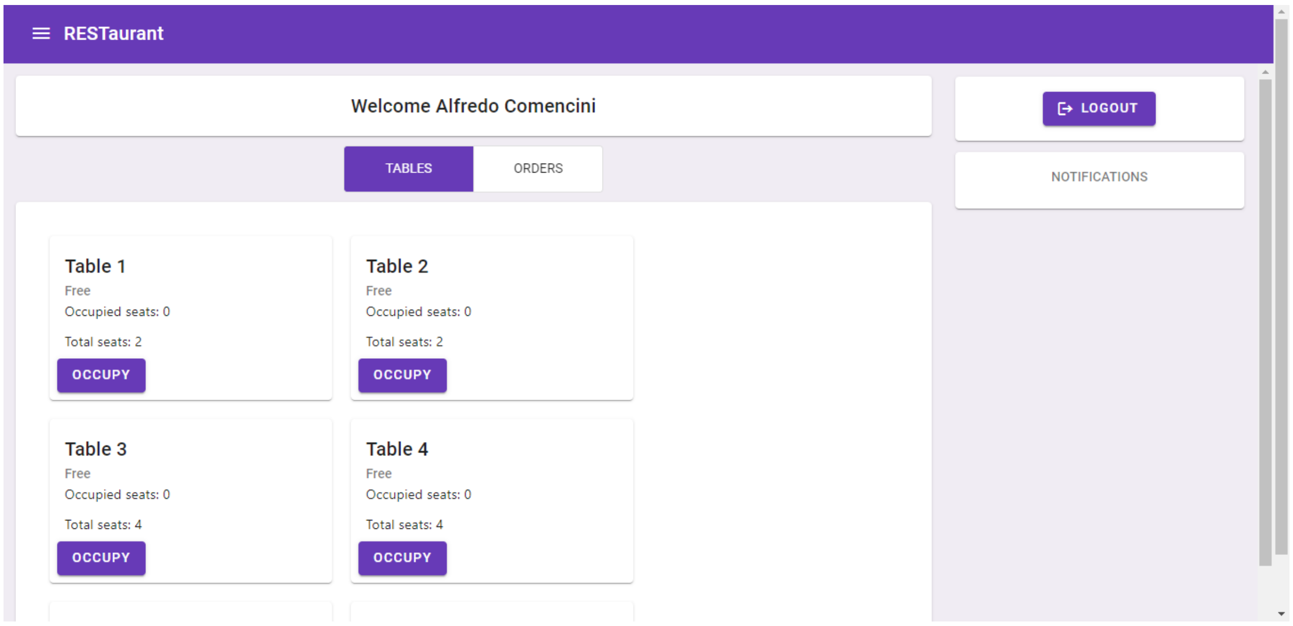
- Add management of the notes of an order, which are currently only present in the database model;
- Add a table reservation field;
- Being able to divide the food/beverages in the menu according to their type, perhaps even implementing a search system based on that type to avoid displaying the entire menu but only a portion of it;

Screenshot

Firstly, a login is required:



We log in as Alfredo, a waiter.
A waiter can occupy/free one or more tables



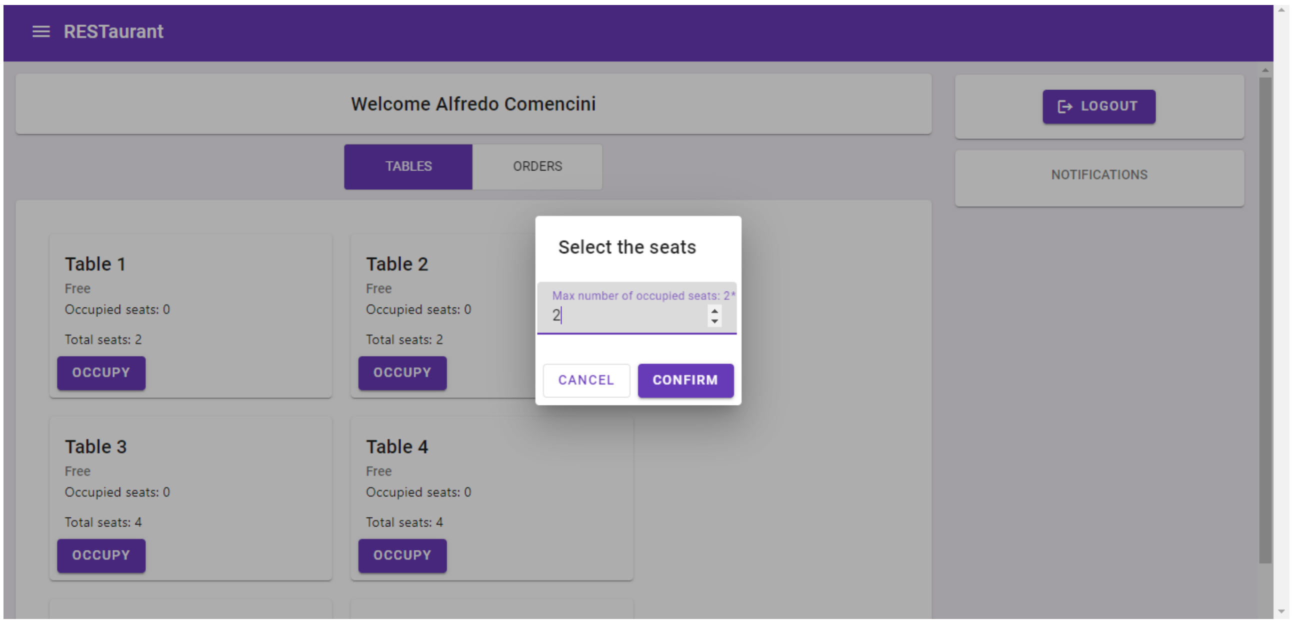
Welcome Alfredo Comencini

Table 1
Free
Occupied seats: 0
Total seats: 2
OCCUPY

Table 2
Free
Occupied seats: 0
Total seats: 2
OCCUPY

Table 3
Free
Occupied seats: 0
Total seats: 4
OCCUPY

Table 4
Free
Occupied seats: 0
Total seats: 4
OCCUPY


RESTaurant

Welcome Alfredo Comencini

LOGOUT

TABLES ORDERS

NOTIFICATIONS

Table 1
Free
Occupied seats: 0
Total seats: 2
OCCUPY

Table 2
Free
Occupied seats: 0
Total seats: 2
OCCUPY

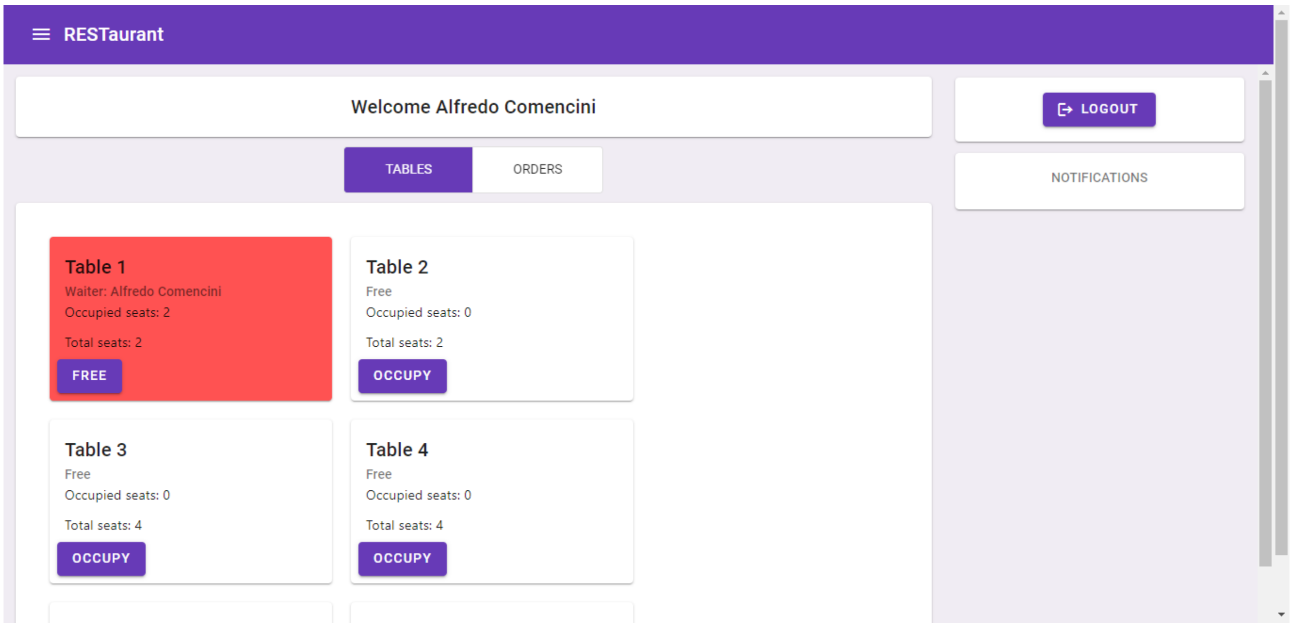
Select the seats

Max number of occupied seats: 2*
2

CANCEL CONFIRM

Table 3
Free
Occupied seats: 0
Total seats: 4
OCCUPY

Table 4
Free
Occupied seats: 0
Total seats: 4
OCCUPY


RESTaurant

Welcome Alfredo Comencini

LOGOUT

TABLES ORDERS

NOTIFICATIONS

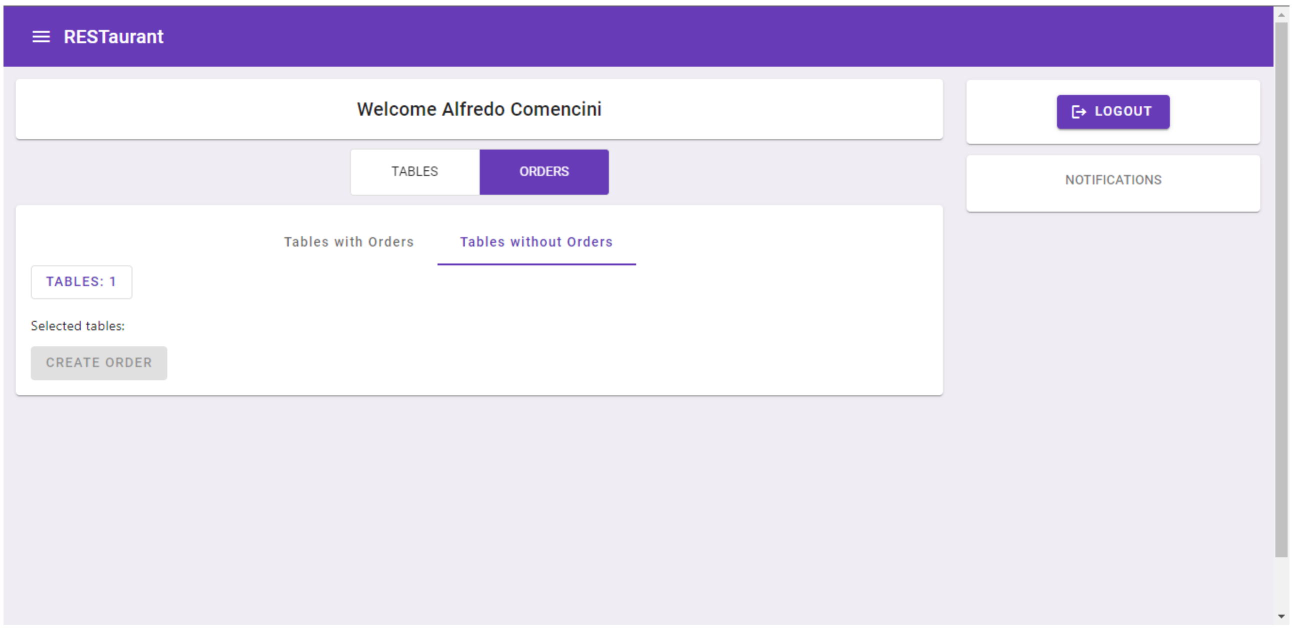
Table 1
Waiter: Alfredo Comencini
Occupied seats: 2
Total seats: 2
FREE

Table 2
Free
Occupied seats: 0
Total seats: 2
OCCUPY

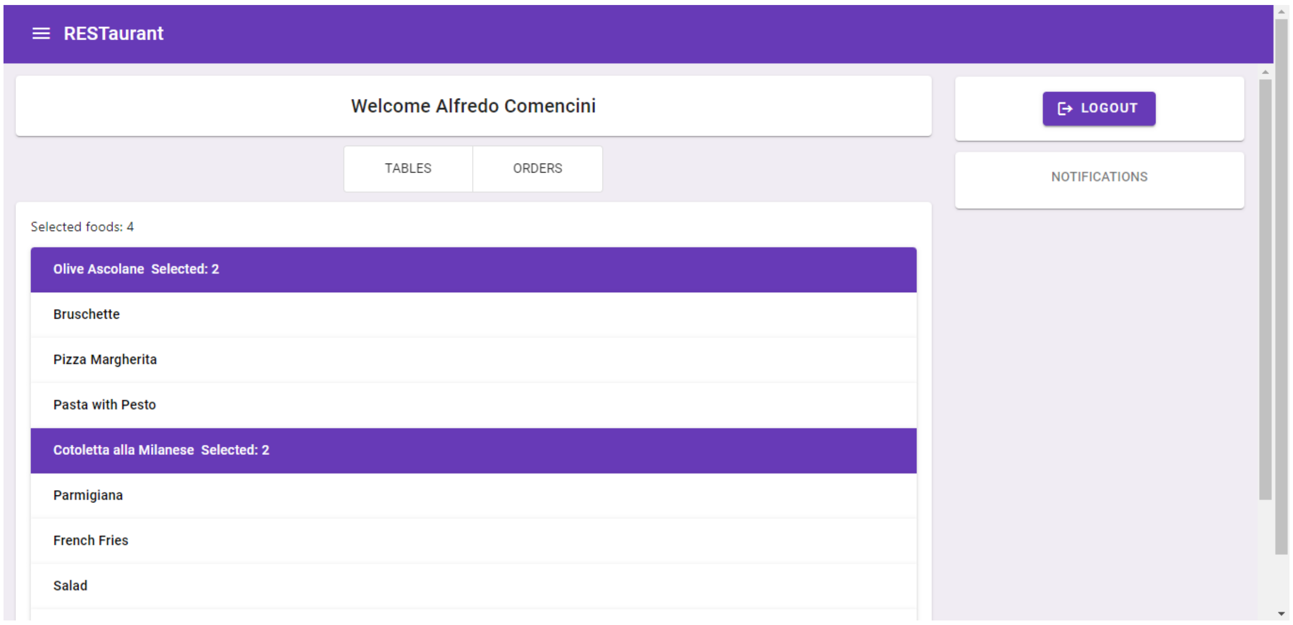
Table 3
Free
Occupied seats: 0
Total seats: 4
OCCUPY

Table 4
Free
Occupied seats: 0
Total seats: 4
OCCUPY

Once he has at least one table to serve, he can create an order by selecting that table.
In case of more tables occupied, he can select more tables to create a joined table

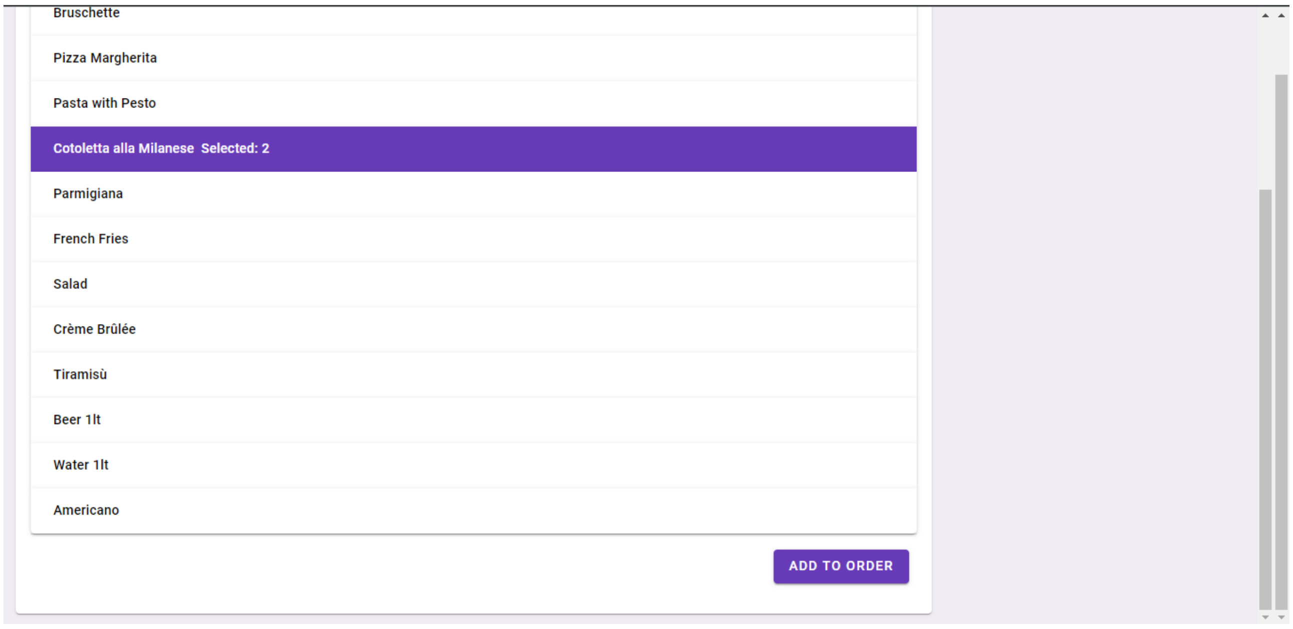


Once at least one table has been selected, the menu is shown and the waiter can take the order

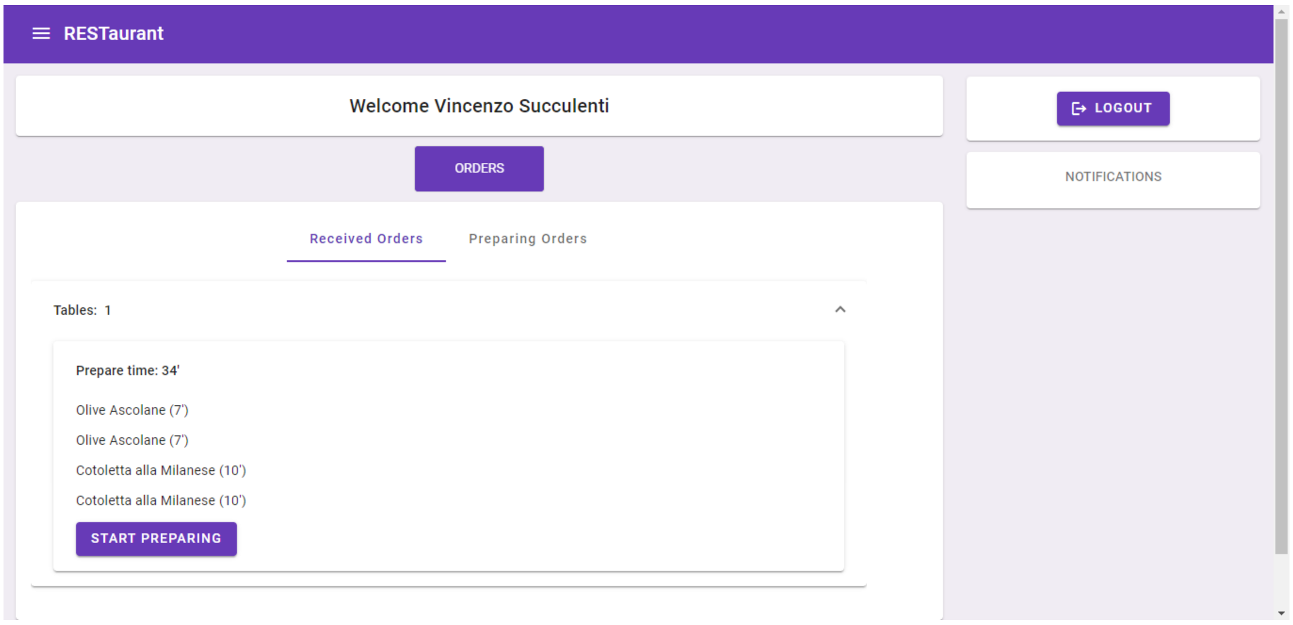


Selected foods: 4

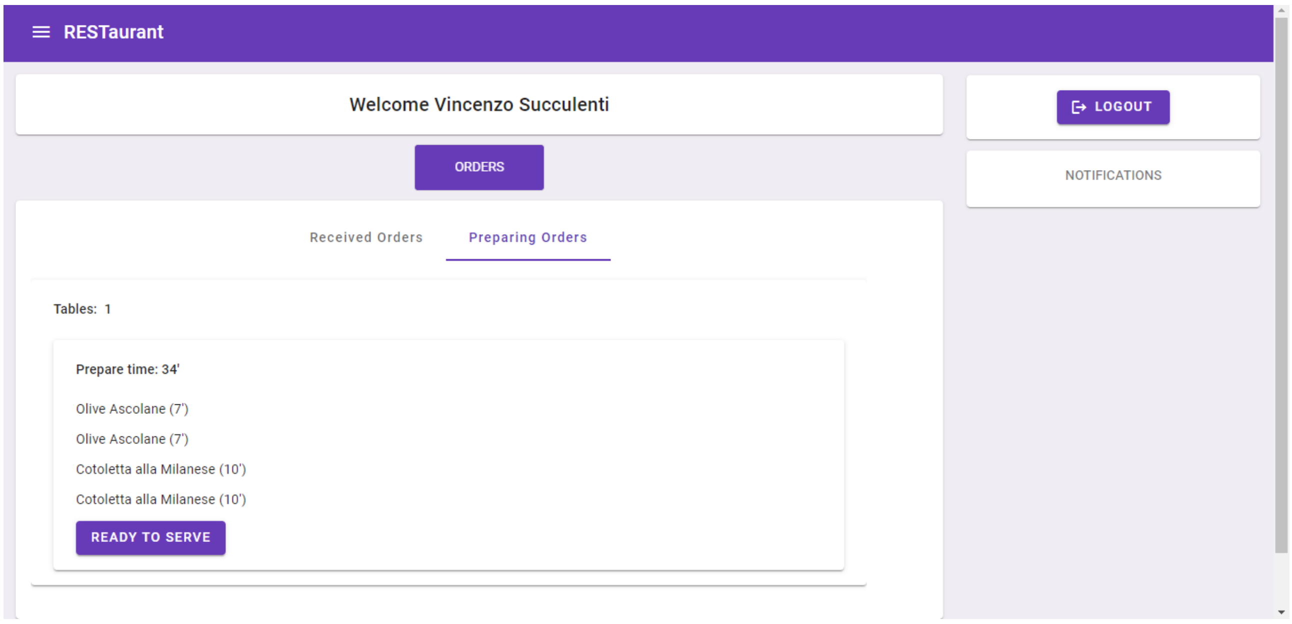
Olive Ascolane Selected: 2
Bruschette
Pizza Margherita
Pasta with Pesto
Cotoletta alla Milanese Selected: 2
Parmigiana
French Fries
Salad


Bruschette
Pizza Margherita
Pasta with Pesto
Cotoletta alla Milanese Selected: 2
Parmigiana
French Fries
Salad
Crème Brûlée
Tiramisù
Beer 1lt
Water 1lt
Americano
ADD TO ORDER

We log out and log in again, this time as Vincenzo, a cook.
A cook, just like a barman, has access to the orders ready to be taken and to the orders he already started preparing.
He can start preparing the order Alfredo just sent


RESTaurant
Welcome Vincenzo Succulenti
LOGOUT
ORDERS
NOTIFICATIONS
Received Orders
Preparing Orders
Tables: 1
Prepare time: 34'
Olive Ascolane (7')
Olive Ascolane (7')
Cotoletta alla Milanese (10')
Cotoletta alla Milanese (10')
START PREPARING

Once he finished cooking, he can notify Alfredo that the order is ready to be served



Tables: 1

Prepare time: 34'

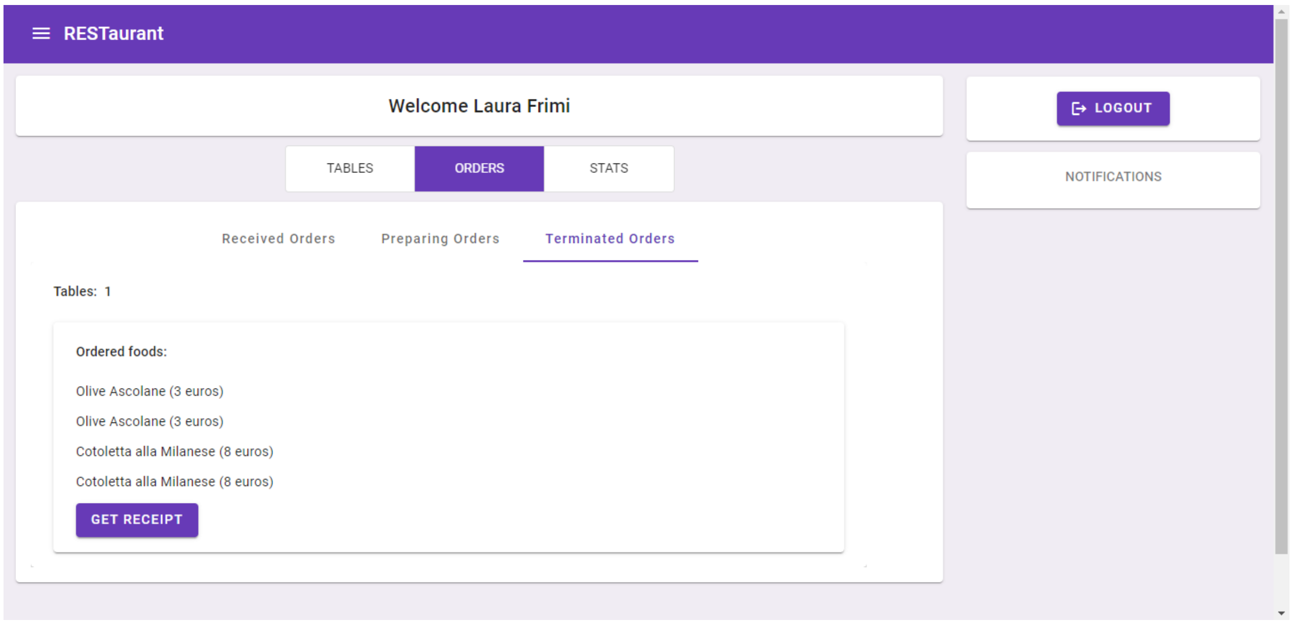
Olive Ascolane (7')

Olive Ascolane (7')

Cotoletta alla Milanese (10')

Cotoletta alla Milanese (10')

We log out and log in again, this time as Laura, a cashier
The cashier has access to all the orders, especially the ones ready to be paid



Tables: 1

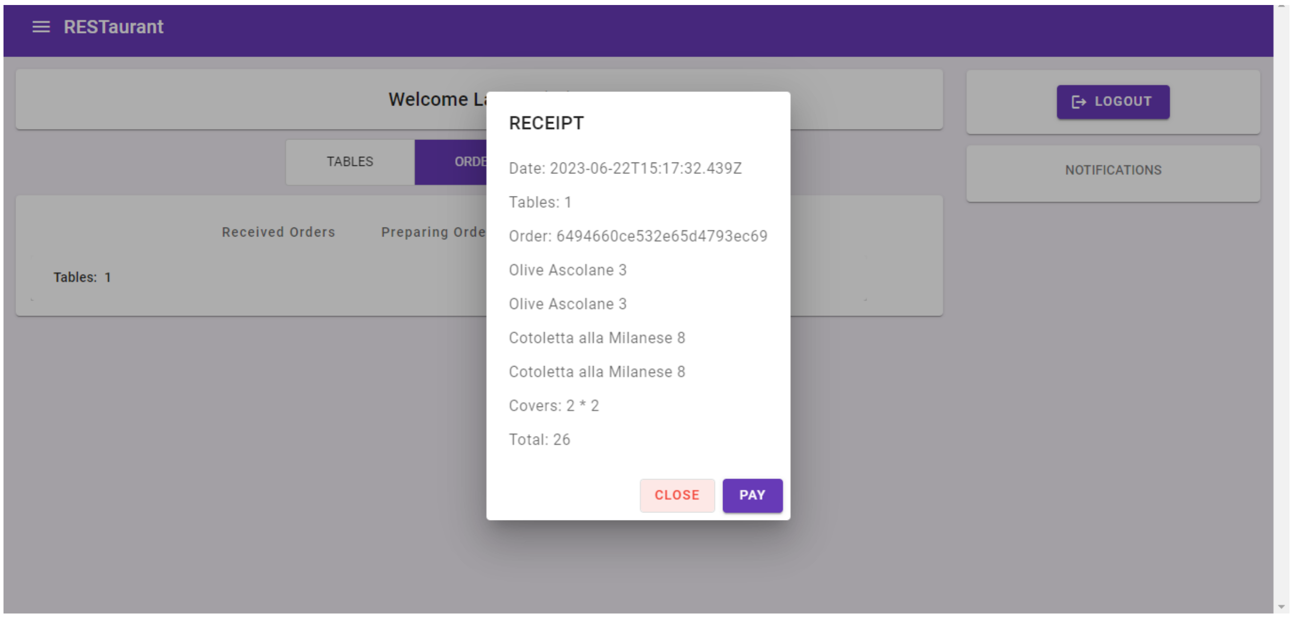
Ordered foods:

Olive Ascolane (3 euros)

Olive Ascolane (3 euros)

Cotoletta alla Milanese (8 euros)

Cotoletta alla Milanese (8 euros)



Tables: 1

RECEIPT

Date: 2023-06-22T15:17:32.439Z

Tables: 1

Order: 6494660ce532e65d4793ec69

Olive Ascolane 3

Olive Ascolane 3

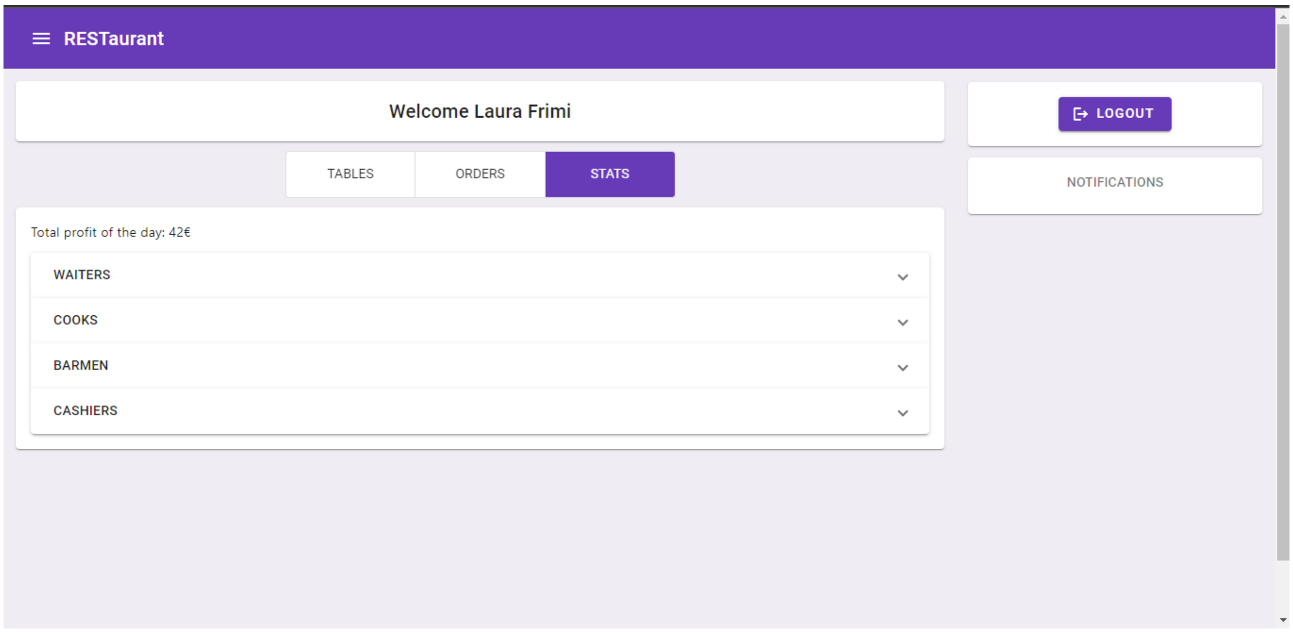
Cotoletta alla Milanese 8

Cotoletta alla Milanese 8

Covers: 2 * 2

Total: 26

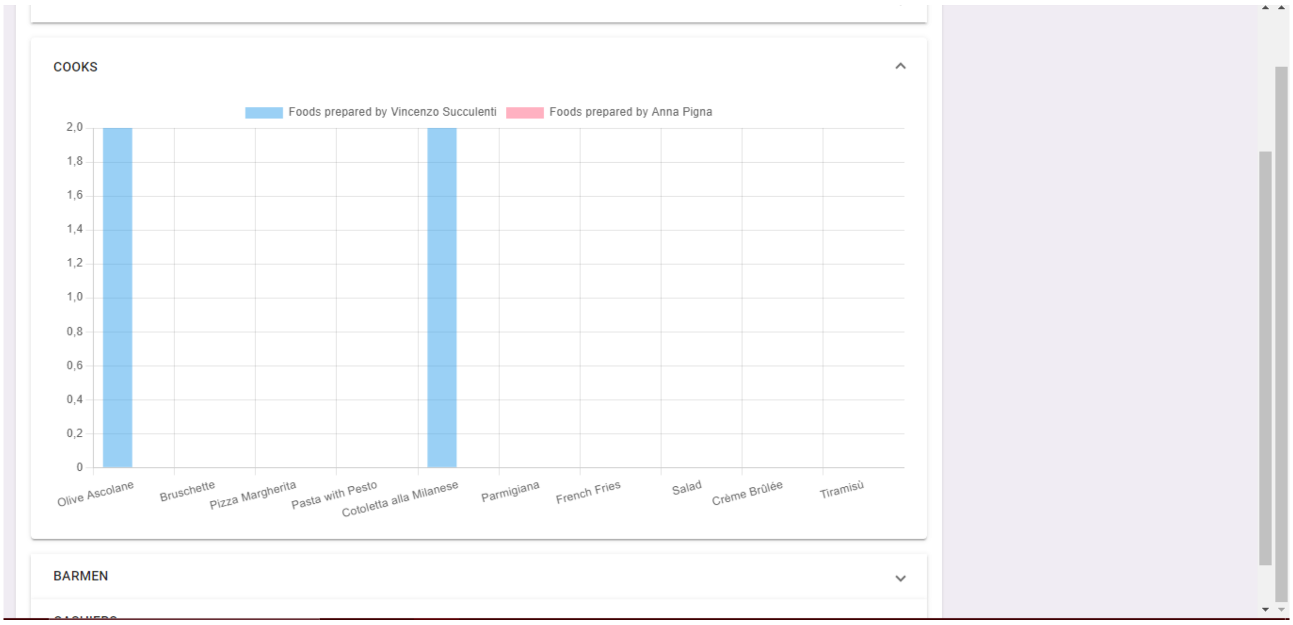
Once the order has been paid, it is no longer available, and the total profit is increased.
The total profit is available in the statistics page, another feature a cashier has access to



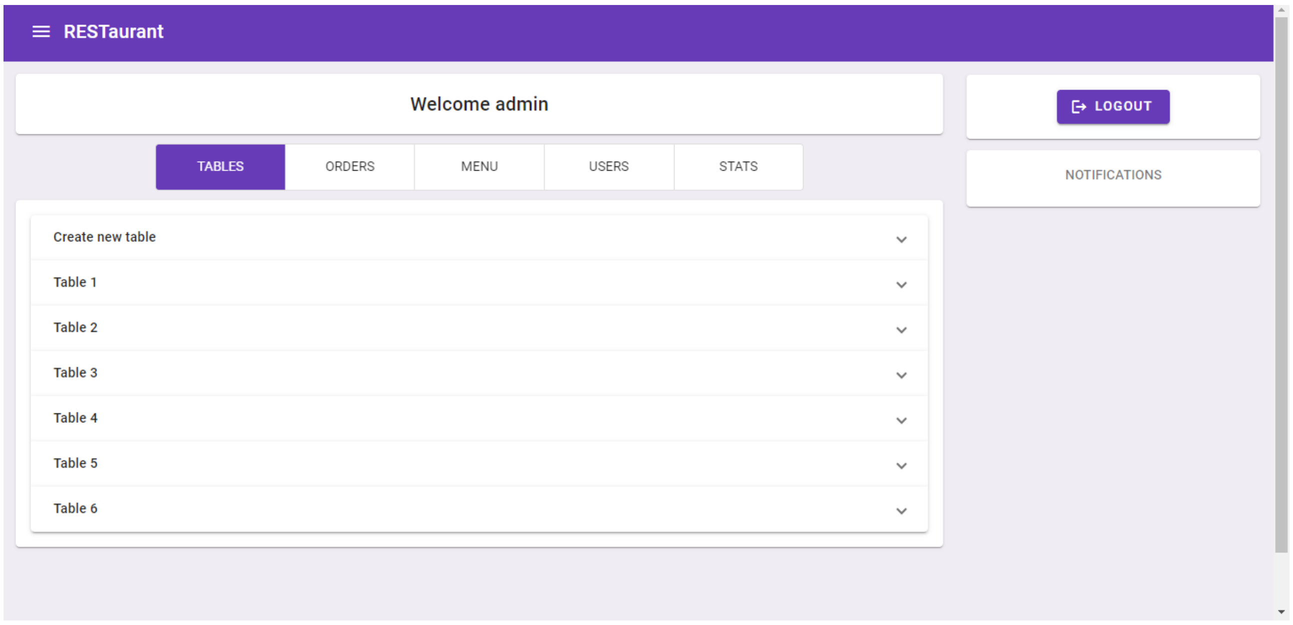
Welcome Laura Frimi

Total profit of the day: 42€

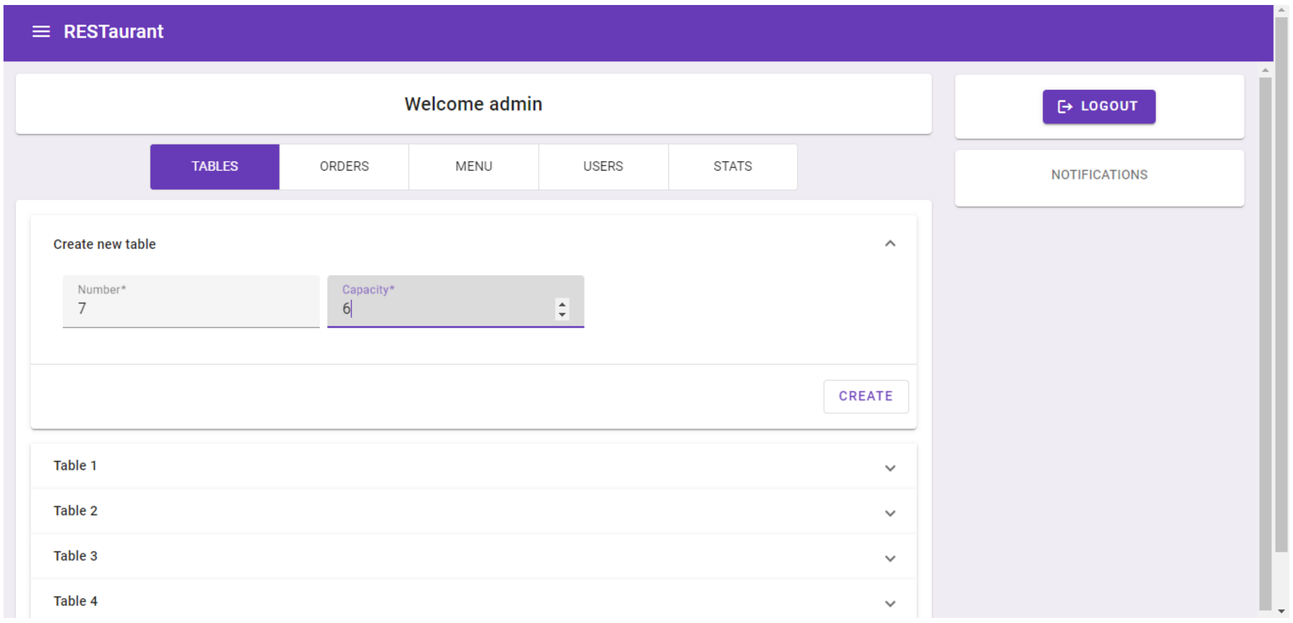
Here the cashier can check some statistics, for example statistics regarding cooks



Finally, we log out and log in again as the admin.
A new table must be added to the restaurant



Create new table
Table 1
Table 2
Table 3
Table 4
Table 5
Table 6


RESTaurant

Welcome admin

TABLES ORDERS MENU USERS STATS

LOGOUT

NOTIFICATIONS

Create new table

Number* 7

Capacity* 6

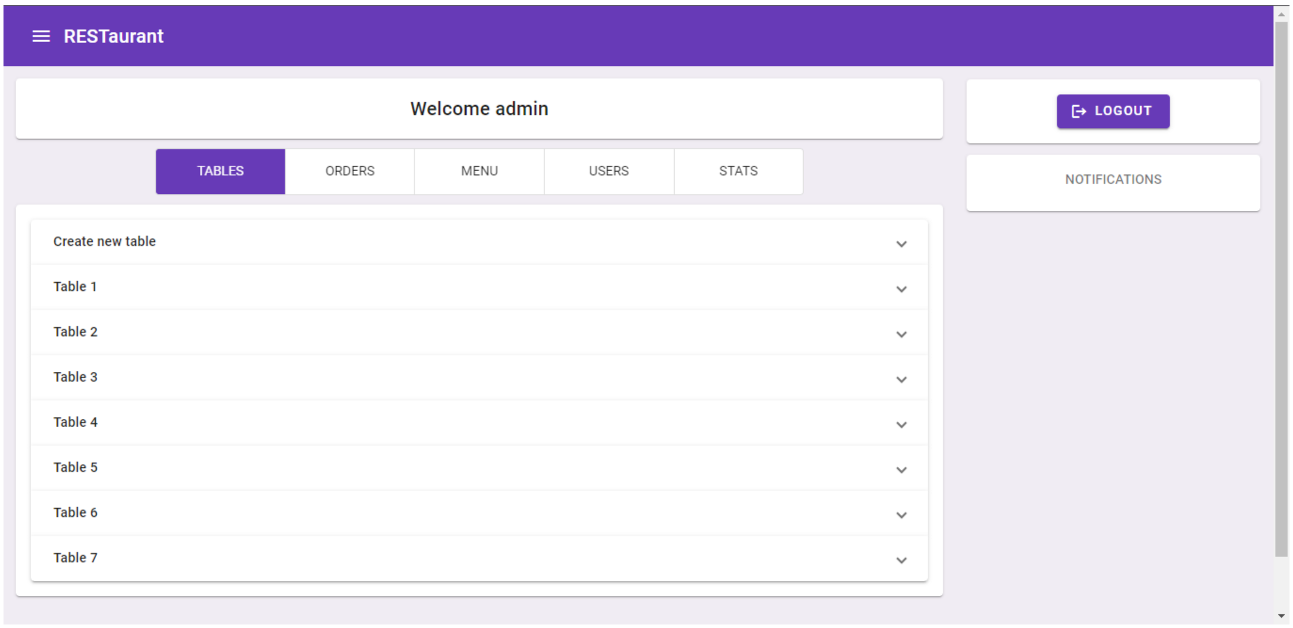
CREATE

Table 1

Table 2

Table 3

Table 4


RESTaurant

Welcome admin

TABLES ORDERS MENU USERS STATS

LOGOUT

NOTIFICATIONS

Create new table

Table 1

Table 2

Table 3

Table 4

Table 5

Table 6

Table 7