Università
Ca'Foscari
Venezia

Course of Tecnologie e Applicazioni Web

Exam project report

RESTaurant

Professor
Prof. Filippo Bergamasco

Student
Martino Pistellato
Matricola 886493

Group
Luca Furegon, 886933
Martino Pistellato, 886493

Academic Year
2022 / 2023

Contents

List of Figures

List of Codes

List of Tables

Chapter 1

Introduction

The project consists of a RESTful web application to manage the needs of a restaurant; from here the name: RESTaurant. This app allows to create orders, occupy or free tables, manage the preparation queues, access statistics about the workers and other options that will be shown in the next sections. This project has been made in collaboration with my colleague Luca Furegon. The primary programming language employed for this project is TypeScript, with MongoDB serving as the chosen database. Additionally, the project utilizes the Express framework, as well as Angular and React frameworks.

Chapter 2

Architecture

This web app is divided into two main parts: backend and frontend. The former is the server and consists of both the setup and the connection to the database, offering through a RESTful API data to the client; the latter utilizes the data received from the server to provide a responsive interface to manage and support all the operations that are needed in a restaurant. Both backend and frontend will be analyzed in the following sections.

2.1 Backend

As said before, the backend is divided in two parts: the database related one and the routing/API one. Starting from the first, the system relies on a connection to an online MongoDB non-relational database: it first connects to the cluster, then it searches if the database has already been initialized and if true it closes the connection, otherwise it creates the dabtabase (simply by establishing a connection to it), the collections and it fills them with some default data. In this case it creates some users, tables and foods. This section can be changed in order to meet the needs of the owner of the restaurant; the default foods and tables can be easily modified, as well as the users that get created. Currently, the space where the database is created and managed is private and accessible only through credentials known by my colleague and me; again, this is easily changeable because the owner of the restaurant can create his own space and then substitute our development connection with his personal link.

2.1.1 Data models

Here all of the data models that constitute the database are analyzed: it will be shown both the relative code and the saved object; how to modify, add or delete these objects will be described in section 2.1.2.

User

This class represents the users and it determines not only what he is capable of, but also how much work he has done in his current shift. First of all, the role of a user is coded as an enumeration, where the possible values are the possible roles; a user can have only one role at a time and currently five roles are set:

- Admin, can freely interact with the database and create/modify/delete almost everything. It can also access the statistics about the workers.
- Cashier, can see the preparation queues and how many orders are currently received, being prepared or terminated and ready to be payed. it can also access the statistics.

- Barman, can only see the orders containing drinks that are received or that are being prepared by him. Gets notified as soon as a new order is made.
- Cook, the same as the barman, only with foods instead of drinks.
- Waiter, can free/occupy tables and take orders. It gets notified every time an order relative to one of the table that he is currently serving is ready.

A user is not only made of a role, but also a name, an email and password to implement the login and avoid confusions and an array called `totalWorks`. It contains all the works that the user has done in his current shift: a cook or barman will have all the foods that he has prepared and a cashier all the receipt that he has payed. Waiters work differently because their statistics consists of how many tables or clients are serving in a specific moment. This array gets emptied every time the user logs in. The structure of a user in the database is shown in figure 2.1. The user also has two different methods: one for setting, encrypting and saving the password when modified, one to verify the password in the login phase.

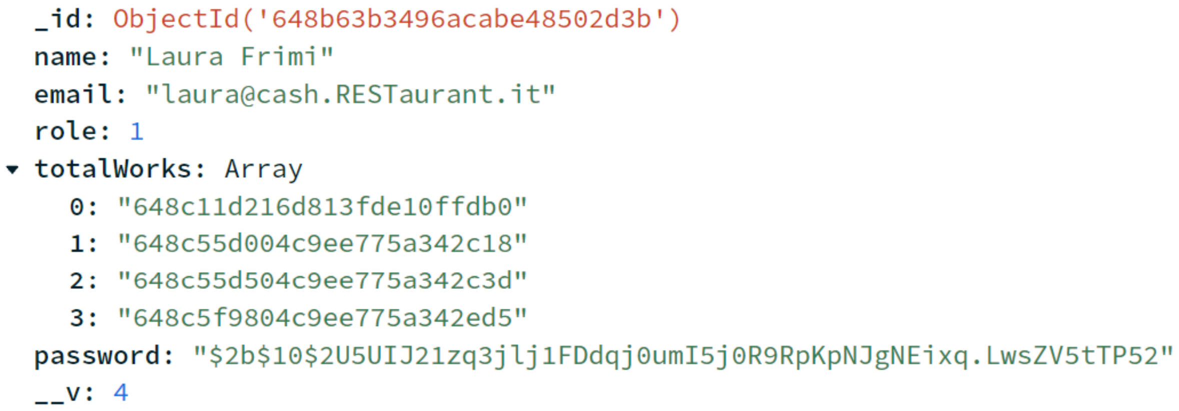
It is worth noting that as the user logs in, he gets associated with a token which allows him to authenticate through the app. This token has a lifespan of up to 9 hours, which is a standard day of work with an hour of launch break. Everything related to the token will be discussed more in depth in the next sections.

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

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

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

userSchema.methods.setPassword = function(pwd: string) {
    this.password = bcrypt.hashSync(pwd, 10);
}

userSchema.methods.validatePassword = function(pwd: string): boolean {
    return bcrypt.compareSync(pwd, this.password);
}
```

Code 2.1: User model definition

Table

In this work, a table is composed of a `table_number` that represent the table and is unique, two numbers that indicate the total capacity and the current number of people occupying that table, a true/false field to easily understand if it is free or not, a reference to the waiter who occupied the table and must serve it and lastly an array of tables if the waiter had to join



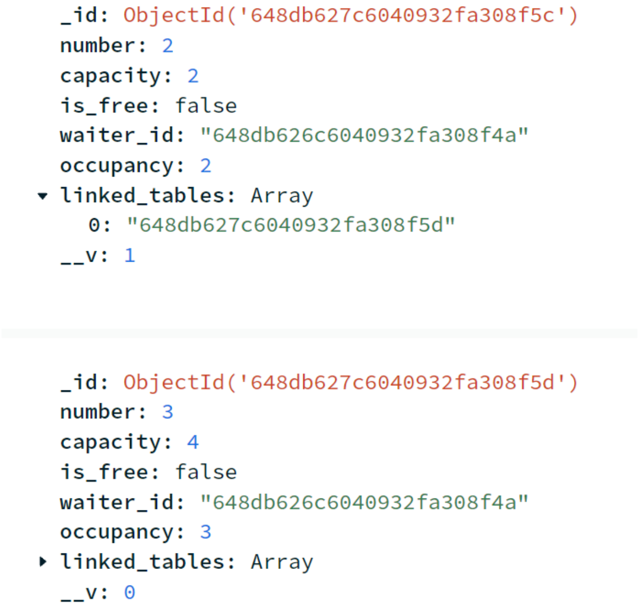
```
  _id: ObjectId('648b63b3496acabe48502d3b')
  name: "Laura Frimi"
  email: "laura@cash.RESTaurant.it"
  role: 1
▾ totalWorks: Array
    0: "648c11d216d813fde10ffdb0"
    1: "648c55d004c9ee775a342c18"
    2: "648c55d504c9ee775a342c3d"
    3: "648c5f9804c9ee775a342ed5"
  password: "$2b$10$2U5UIJ21zq3jlj1FDdqj0umI5j0R9RpKpNJgNEixq.LwsZV5tTP52"
  __v: 4
```

Figure 2.1: Example of a user in the database

multiple tables to create the space for a large group of customers. If two or more tables are linked, only the main one (which is the first that gets selected) will contain the array with the references to the other tables; an example is shown in figure 2.2. There is also a method which is used to free/occupy a specific table and the commented field will be discussed in chapter 5.

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

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

Code 2.2: Table model definition



```
  _id: ObjectId('648db627c6040932fa308f5c')
  number: 2
  capacity: 2
  is_free: false
  waiter_id: "648db626c6040932fa308f4a"
  occupancy: 2
▾ linked_tables: Array
    0: "648db627c6040932fa308f5d"
  __v: 1
```

```
  _id: ObjectId('648db627c6040932fa308f5d')
  number: 3
  capacity: 4
  is_free: false
  waiter_id: "648db626c6040932fa308f4a"
  occupancy: 3
▸ linked_tables: Array
  __v: 0
```

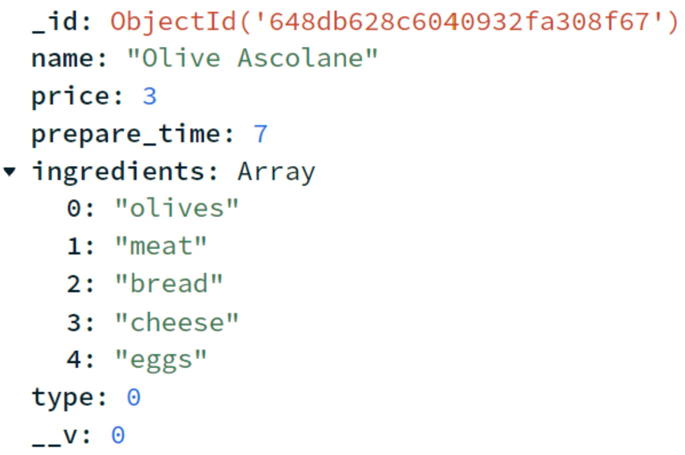
Figure 2.2: Example of two linked tables in the database

Food

The food model is quite simple: it contains the name, price, preparation time (in minutes) and ingredients of the food. It also includes a type described by the enumeration `foodTypes`. Its main purpose is to distinguish between solid foods and drinks, although this use could be expanded as will be discussed in chapter 5. An example is shown in figure 2.3.

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

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

Code 2.3: Food model definition



```
 _id: ObjectId('648db628c6040932fa308f67')
 name: "Olive Ascolane"
 price: 3
 prepare_time: 7
▾ingredients: Array
   0: "olives"
   1: "meat"
   2: "bread"
   3: "cheese"
   4: "eggs"
 type: 0
 __v: 0
```

Figure 2.3: Example of a food in the database

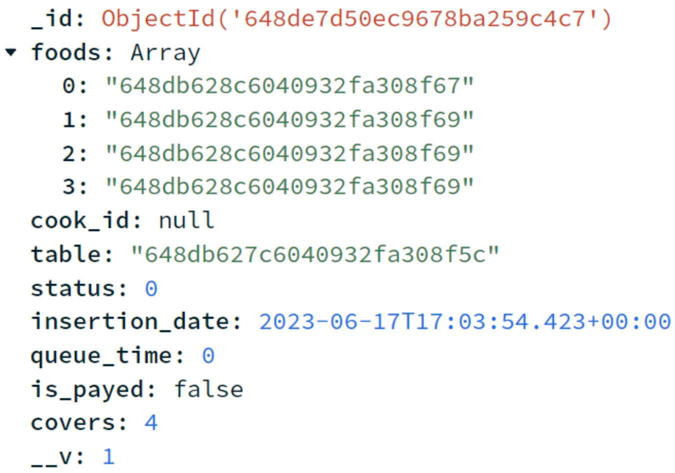
Order

The order model sums up everything seen so far: the ordered foods are stored in the relative array, the cook who decided to start preparing it is saved, the main table related to the order in stored as well as the number of clients seated and the total time needed to prepare it; there is also an insertion date, a boolean that indicates if the order has been payed or not and a field that shows in which status the order is. Again, the commented field `notes` will be discussed in chapter 5.

Instead of having one big single order that gets modified, multiple orders can be made while all referring to the same table. This way, if a table first orders the drinks and only after orders the foods, two different orders will be created. Even though it doesn't seem so, managing multiple orders is easier for the cooks (or barmen) because if someone start preparing the relative food and the order gets modified, there's no way to distinguish between what has been done until that moment. You could create more arrays to store the different foods in the different stages of preparation, but what if a different cook from the first one starts preparing what has been added to the order. It's difficult to track who did what, so by splitting the order into smaller ones it becomes much easier and an example can be seen in figure 2.4.

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

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
}
});
```

Code 2.4: Order model definition



```
_id: ObjectId('648de7d50ec9678ba259c4c7')
▾ foods: Array
    0: "648db628c6040932fa308f67"
    1: "648db628c6040932fa308f69"
    2: "648db628c6040932fa308f69"
    3: "648db628c6040932fa308f69"
  cook_id: null
  table: "648db627c6040932fa308f5c"
  status: 0
  insertion_date: 2023-06-17T17:03:54.423+00:00
  queue_time: 0
  is_payed: false
  covers: 4
  __v: 1
```

Figure 2.4: Example of an order in the database

2.1.2 REST APIs

The API follows the RESTful style and its analysis will be divided in different parts: one for each model plus one for the login.

Login

Endpoint	Method	Parameters	Description
/login	GET	email and password	Returns a signed json-web-token if the authentication succeeds

Table 2.1: Login specific API

User

Endpoint	Method	Parameters	Description
/users	GET		Returns a list with all the users
/users	POST	name, email, role and password of the new user	Creates a new user with the passed parameters
/users/:userID	PUT	userID is mandatory, while name, email, role and password can be null	Updates the user identified by the given id
/users/:userID	DELETE	userID	Deletes the user identified by the given id

Table 2.2: User specific API

Food

Endpoint	Method	Parameters	Description
/foods	GET		Returns a list with all the foods
/foods	POST	name, price, prepareTime, type, ingredients	Creates a new food with the passed parameters
/foods/:foodID	DELETE	foodID	Deletes the food identified by the given id
/foods/:foodID	PATCH	foodID is mandatory, name, price, prepareTime, type and ingredients may be null	Updates the selected food if the parameters sent are not null

Table 2.3: Food specific API

Table

Endpoint	Method	Parameters	Description
/tables	GET		Returns a list with all the tables
/tables/serving	GET		Returns a list with all the tables served by a specific waiter
/tables/:tableID	GET		Returns specific table, identified by the id
/tables	POST	table number, capacity	Creates a new table with the passed parameters
/tables/:tableID	DELETE	tableID	Deletes the table identified by the given id
/tables/	PUT	tableID, waiterID, occupancy	Calls the function `changeStatus()` on the table identified by the given id: it is used either to occupy a table with the given occupancy and link it to the waiter, or to free it and clear those fields
/tables/link	PUT	Table[ ]	Links the tables to create a virtual singular table: it takes the first table of the array and fills its field `linked_tables` with the remaining ones
/tables/:tableID	PATCH	tableID is mandatory, capacity and number may be null	Updates the selected table if the parameters sent are not null

Table 2.4: Table specific API

Order

Endpoint	Method	Parameters	Description
/orders	GET		Returns a filtered list with all the orders, the filter depends on the role of the user
/orders/all	GET		Returns a list with all the orders, it is used for statistics
/orders/receipt/:orderID	GET	orderID	Creates the receipt for a specific table, merging all the single orders into one
/orders/totalprofit	GET		Gets the total profit earned in a day
/tables/:orderID	DELETE	orderID	Deletes the order identified by the given id. It can be used only by the admin if necessary
/tables/old	DELETE		Deletes the payed orders older than two weeks
/orders/:orderID	PUT	orderID	Updates the selected order, the functioning changes based on the role
/orders/	POST	table	Creates a new order linked to the given table

Table 2.5: Order specific API

2.2 Frontend

The frontend side of the application is based on Angular and its core structure, which consists of components (which define what the client can see and interact with) and services (which let the frontend communicate with the backend), is the following:

```
.
|-- components
|   |-- foods-component
|   |-- home-component
|   |-- login-component
|   |-- orders-component
|   |-- receipt-dialog-component
|   |-- stats-component
|   |-- table-occupancy-dialog-component
|   |-- tables-component
|   |-- users-component
|-- services
    |-- foods-services
    |-- orders-services
    |-- socket-services
    |-- tables-services
    |-- users-services
```

Starting from the components, there is one for each data model or route; the access to them is constrained by a mandatory login, in which the role will be defined and based on that the interface will change, even in the same module. A brief explanation will now be given:

- `foods-component`, it is only accessible or by waiters to add foods to an order, or by admin to update the menu.

- `home-component`, as it will be shown in 4, the home is the core of the entire app: here the other components are loaded and here it is possible to navigate through the application.

- `login-component`, is the first component that the user interacts with: it is mandatory and cannot be evaded.

- `orders-component`, based on the role, here different items are displayed: a waiter will see the tables that he is serving and by selecting them he can proceed and select the food to add to the order, cooks and bartender (their role is very similar) will only see both the orders recently arrived that nobody has already started working on and the list of all the orders which he is currently preparing; finally, cashiers and admins here will see the complete status of all the orders and cashiers will be able to create receipts and pay.

- `stats-component`, this component is accessible by cashiers and admins; here will be displayed all the graphs and statistics about the workers: for the waiters it will be shown how may tables and clients are serving at the moment, for cooks or barman how many of each food have prepared during their shift and for cashiers how much money they made the restaurant earn by making pay the receipts.

- `tables-component`, everybody except for cooks and barmen can access this module and here waiters can occupy a table (or free it if there aren't orders linked to it), admin can either delete, free, update them or create new tables, the cashier can only see the status and which table is being served by who.

- `users-component`, accessible only by admins, this component let them manage the users and update them, delete them or create new workers.
- `receipt-dialog-component`, this is just a pop-up which will contain the receipt with all the orders linked to a specific table.
- `table-occupancy-dialog-component`, another pop-up that lets the waiter indicate how many clients are occupying a table.

The services are used by the components to communicate with the backend through the API analyzed before. A service that is worth noting is `socket-service`, which wraps the Socket.io-client methods and guarantee a responsive communication with the server, which can now (thanks to the middelware Socket.io) notify to the client when something occurs (for example a new order has been placed) so that it can update immediately the content shown to the user.

Chapter 3

Main features

This web application supports all the needs of a restaurant and also adds security functions, all with an attractive interface that works perfectly regardless of device or screen size: whether the user is accessing from mobile or tablet or computer, the interface adapts accordingly. It is also easy to maintain as adding or removing roles and functions is achievable due to the modularity of the code. We will now focus on the main features of the app.

3.1 Authentication and Authorization

The whole application runs (both backend and frontend) in https, so the exchange of data between client and server is secured thanks to the certificates that in this case are self-signed and not trusted by the browsers, but they can easily be changed with authenticated ones. As soon as the user gets created, its password gets encrypted through bcrypt before being saved in the database. After that, the user can login and a token containing its data is signed and associated to the session, as it's mandatory to do in order to gain access to the app. Every time the client makes a request to the server, the token gets verified and if the result is positive, his role is extracted from it and the authorization can happen. This mechanism is shown in the following code:

```
//used to authorize the user from the token
export const my_authorize = (roles: roleTypes[] = []) => {
    return [
        auth, //verifies the token
        (req: any, res: any, next: any) => {
            //checks if the user's role is among those allowed
            if (roles.length && !roles.includes(req.auth.role))
                return res.status(401).json({error: true, errormessage: 'Unhautorized'});
            next();
        }
    ];
}

//checks if the credentials are correct, used only in the login phase
passport.use( new passportHTTP.BasicStrategy(
    function(email: string, password: string, done: Function) {
        userModel.findOne({ email: email }).then((user)=>{
            if( !user )
                return done(null,false,{
                    statusCode: 500,
                    error: true,
                    errormessage:"Invalid user"
                });

            if(user.validatePassword(password))
                return done(null, user);

            return done(null,false,{statusCode: 500,
                error: true,
                errormessage:"Invalid password"
            });
        });
    }
));

//Login route
app.get('/login', passport.authenticate('basic', { session: false }),(req, res) => {
    console.log("Login granted. Generating token" );

    userModel.findById(req.user._id).then((logged_user) => {
        logged_user.totalWorks = [];
        logged_user?.save().then((user) => {
            let tokendata = {
                name: req.user.name,
                role: req.user.role,
                email: req.user.email,
                id : req.user._id
            };

            let token_signed = jsonwebtoken.sign(tokendata, process.env.JWT_SECRET, {
                expiresIn: '9h' } ); //sign the token
            res.send({token: token_signed});
        })
    })
})
```

Code 3.1: Authentication and authorization process

3.2 Security

Even though there is a reliable authentication mechanism, errors can occur; both computer-side and human-side. For this reason, the admins of the system have the power to control each access to the application: if a malicious user steals the credential of someone else and try to damage the system, an admin at every point can change the password for that user and completely log him out. He can also completely delete the user, which cause the same logout result. This way, an admin can always protect the system; if this happens the user will be immediately notified, as shown in figure 3.1.

3.3 Roles

Currently there are five roles supported by the system: Admin, Cashier, Cook, Barman and Waiter. They all have different capabilities and interactions and here they will be listed:

- Admin, can have the most complete view of the system and can modify its roots; he can add, modify or delete almost everything, except for the orders which can only be created by the waiters. He can access the page with the statistics about the users and, as said before, he can also manage the users and their credentials.

- Cashier, can oversee the status of the system by looking at the tables (and which are occupied and served by who), orders (which are the received orders, the ones in preparation and the terminated ones), he can also proceed to the checkout and when paying he can free the related tables; he can also access the page with the statistics. We decided to split the admin and the cashier roles because we believed that it wasn't good for the system to have the cashier, which is a worker, with powers over the whole restaurant. Instead, by splitting the roles this way, only the manager of the restaurant can have that power and a cashier can not cause any trouble.

- Cook, he can only see the orders with food in it, so it won't be disturbed by orders containing only drinks. He can take a received order and start preparing it; once this is done, the order gets linked to him and no other cook can take it from him or start preparing again the same dishes. Once the order is fulfilled, he can notify it to the system and the waiter serving the table related to the order will receive an alert saying that the order for that table is ready.

- Barman, its the same as the cook with the difference that he can only see orders with drinks, not with solid foods.

- Waiter, he can see the tables that are free or occupied and can also make the clients take a seat in a free table, by indicating how many seats are now full. Once this is done, the waiter will be linked to the table until they decide to pay and no waiter can free a table that is not linked to him. This way, whenever an order for an assigned table is ready, the waiter receives a notification and can then serve the food.

3.4 Notifications

A notifications system is required if a fast response is needed and in a restaurant that sure is useful. It has already been discussed the role of notifications in this app as well as their functioning, so here they will be displayed and briefly analyzed.

Figure 3.1 shows the notification displayed to the user who has been logged out by force, while figures 3.2 and 3.3 show the admin side of a modification on the user. This way, if there are multiple admins, all of them get notified if there are major changes.

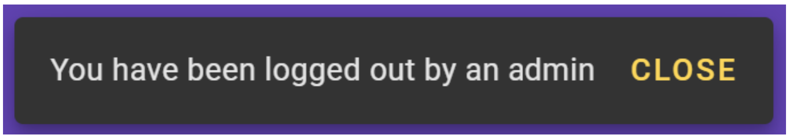

You have been logged out by an admin

Figure 3.1: Push notification in case of a forced logout

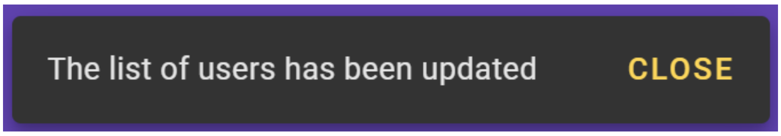
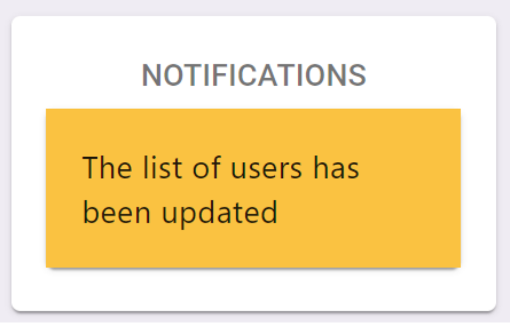

The list of users has been updated

Figure 3.2: Push notification in case of modifications to the user list

Figure 3.4 hows up when a cashier tries to close and pay an order which isn't fully terminated and still has some parts in preparation.


NOTIFICATIONS

The list of users has been updated

Figure 3.3: Side notification in case of modifications to the user list in case of a forced logout

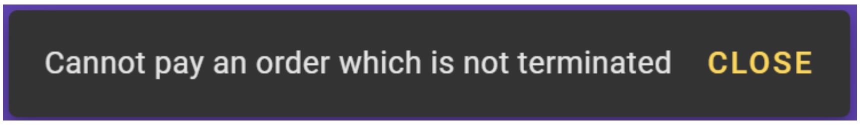

Cannot pay an order which is not terminated

Figure 3.4: Push notification when an order not finished tries to get payed

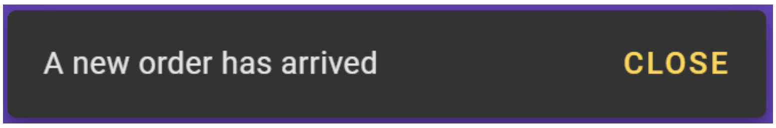
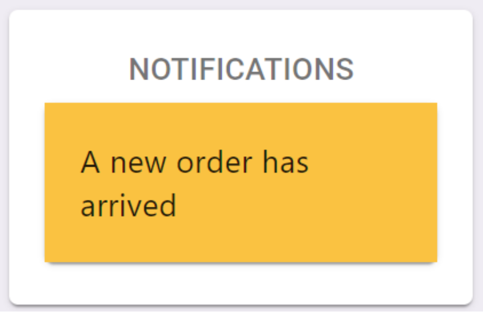

A new order has arrived

Figure 3.5: Push notification in case of the arrival of a new order


NOTIFICATIONS

A new order has arrived

Figure 3.6: Side notification in case of the arrival of a new order

Figures 3.5 and 3.6 show the notification cook or barman side when a new order arrives: they receive a push notification and also one in a side panel that gets canceled only when clicked on.

Lastly, figures 3.7 and 3.8 show the notifications for waiters when a new order linked to them gets prepared and is ready to be served.

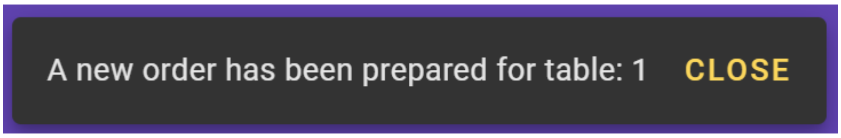
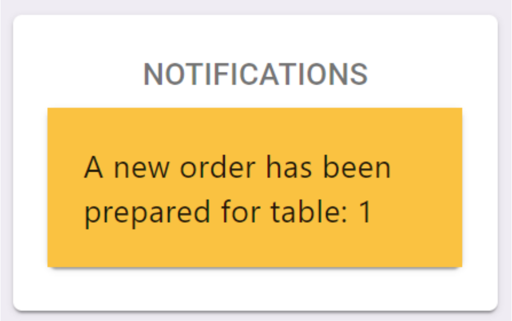

A new order has been prepared for table: 1

Figure 3.7: Push notification in case of the preparation of an order



NOTIFICATIONS

A new order has been prepared for table: 1

Figure 3.8: Side notification in case of the preparation of an order

Chapter 4

Interface

In the following chapter some screenshots and images of the interface will be loaded; each figure will have its own description explaining briefly what is shown.

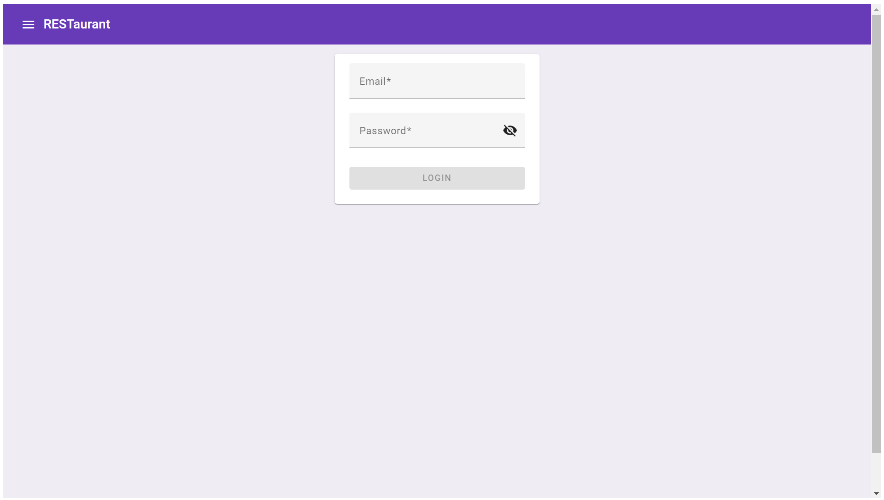


Figure 4.1: Login page (Desktop)

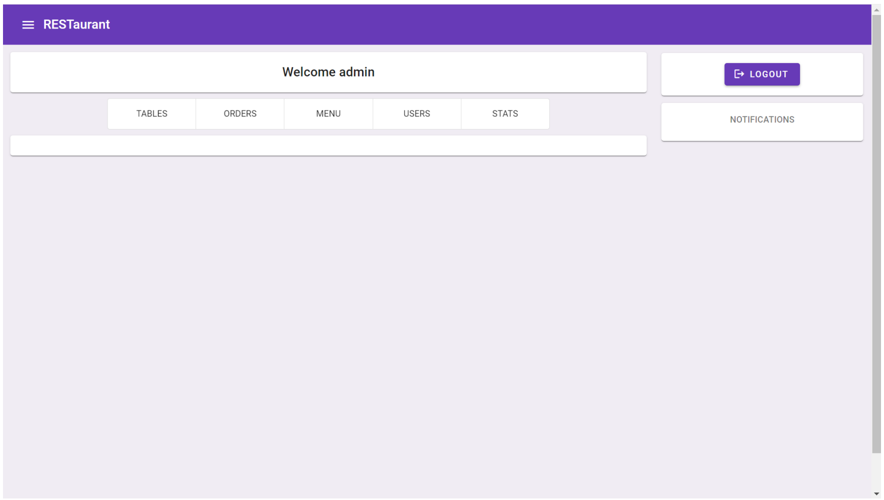
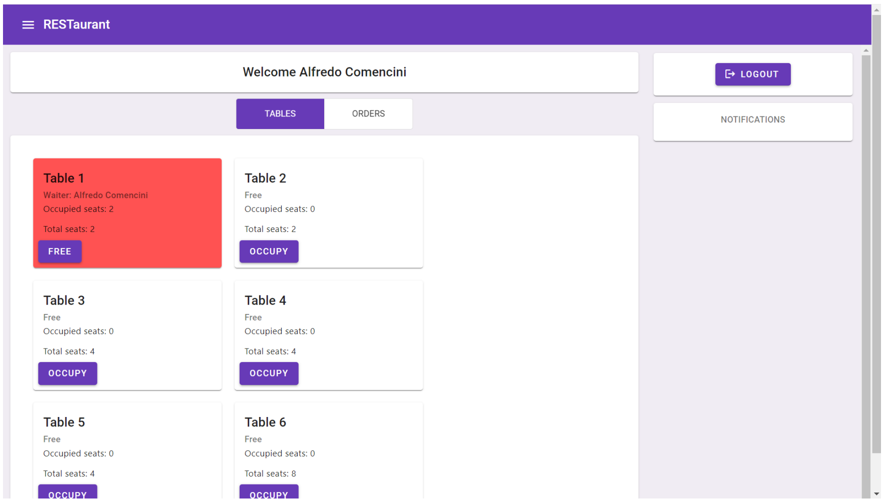


Figure 4.2: Home page with all possibilities for an admin (Desktop)



Welcome Alfredo Comencini

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

Table 2
Free
Occupied seats: 0
Total seats: 2

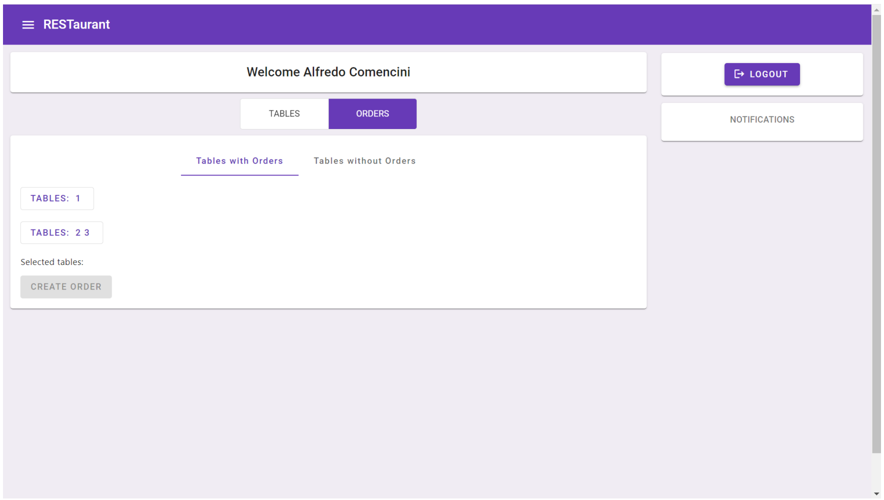
Table 3
Free
Occupied seats: 0
Total seats: 4

Table 4
Free
Occupied seats: 0
Total seats: 4

Table 5
Free
Occupied seats: 0
Total seats: 4

Table 6
Free
Occupied seats: 0
Total seats: 8

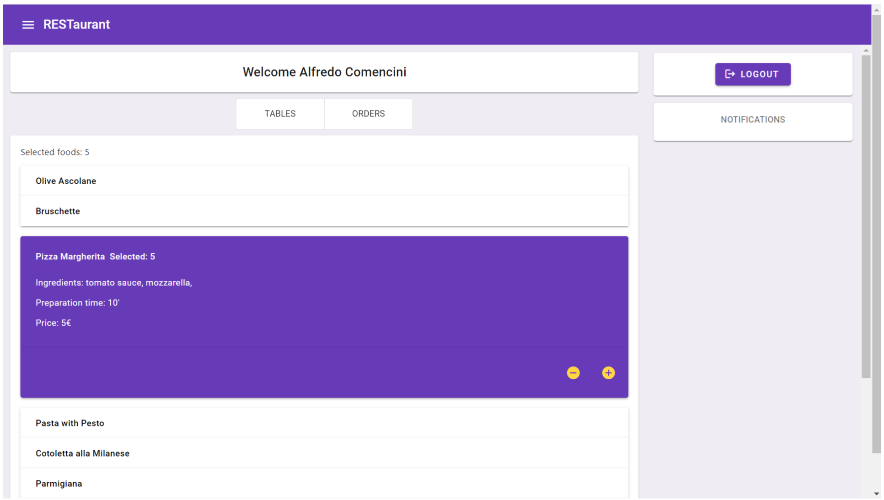
Figure 4.3: A waiter has just occupied the first table (Desktop)



Welcome Alfredo Comencini

Selected tables:

Figure 4.4: The waiter is now selecting the table that wants to make an order (Desktop)



Welcome Alfredo Comencini

Selected foods: 5

Olive Ascolane

Bruschette

Pizza Margherita Selected: 5

Ingredients: tomato sauce, mozzarella,

Preparation time: 10'

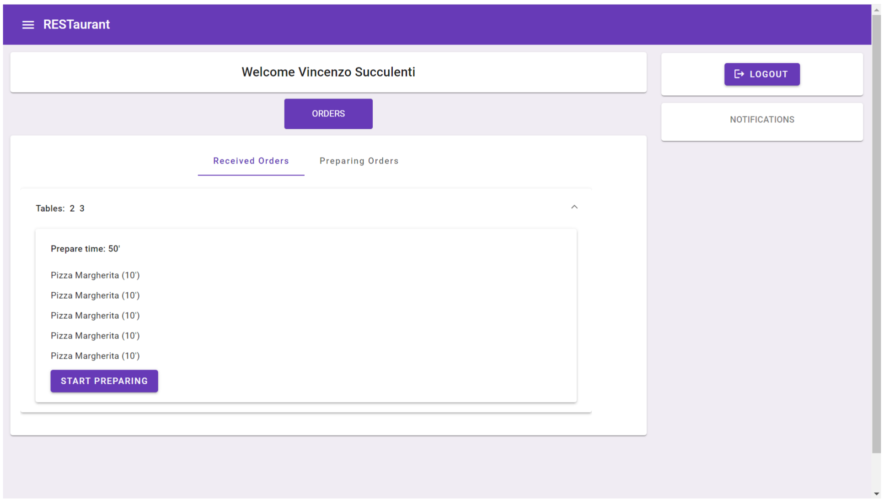
Price: 5€

Pasta with Pesto

Cotoletta alla Milanese

Parmigiana

Figure 4.5: Now the waiter can select the food to add to the order (Desktop)



Welcome Vincenzo Succulenti

Tables: 2 3

Prepare time: 50'

Pizza Margherita (10')

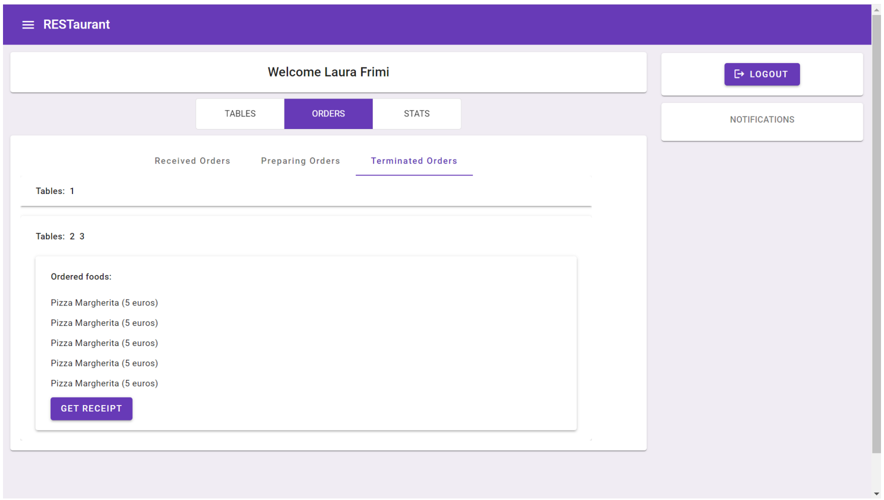
Pizza Margherita (10')

Pizza Margherita (10')

Pizza Margherita (10')

Pizza Margherita (10')

Figure 4.6: Visualization of the order queue by a cook (Desktop)



Welcome Laura Frimi

Tables: 1

Tables: 2 3

Ordered foods:

Pizza Margherita (5 euros)

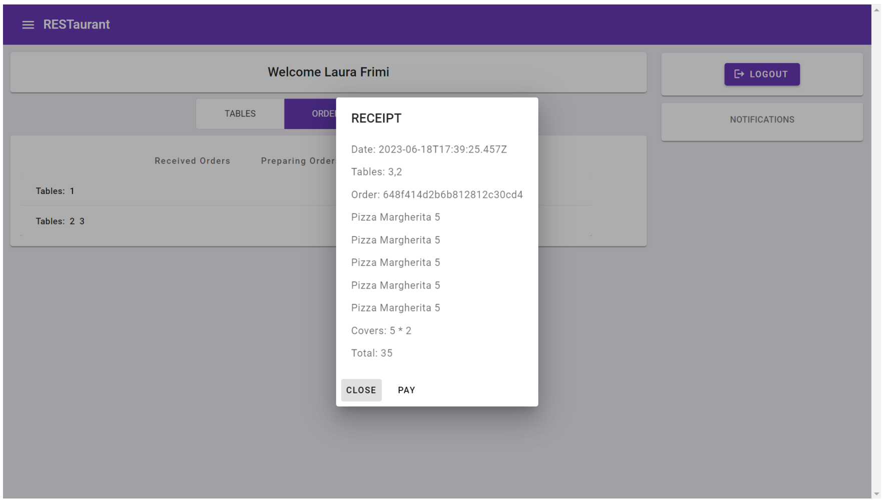
Pizza Margherita (5 euros)

Pizza Margherita (5 euros)

Pizza Margherita (5 euros)

Pizza Margherita (5 euros)

Figure 4.7: Visualization of the terminated orders by a cashier (Desktop)



Welcome Laura Frimi

Tables: 1

Tables: 2 3

RECEIPT

Date: 2023-06-18T17:39:25.457Z

Tables: 3,2

Order: 648f414d2b6b812812c30cd4

Pizza Margherita 5

Pizza Margherita 5

Pizza Margherita 5

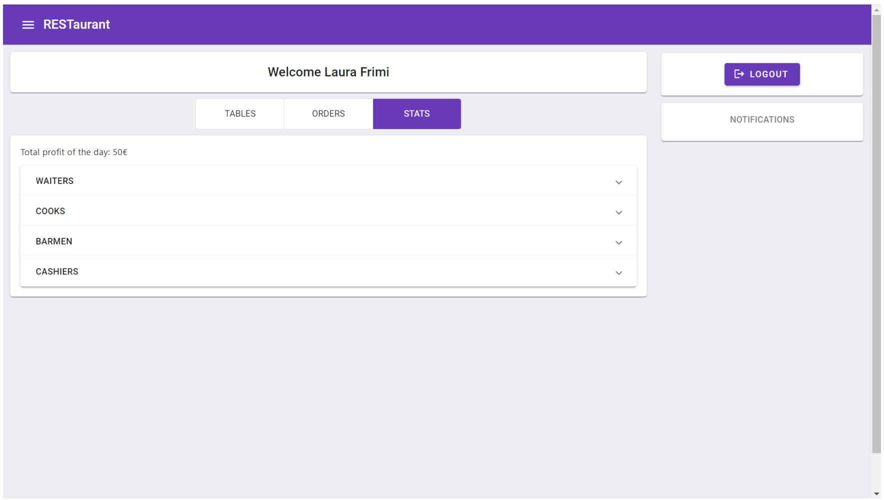
Pizza Margherita 5

Pizza Margherita 5

Covers: 5 * 2

Total: 35

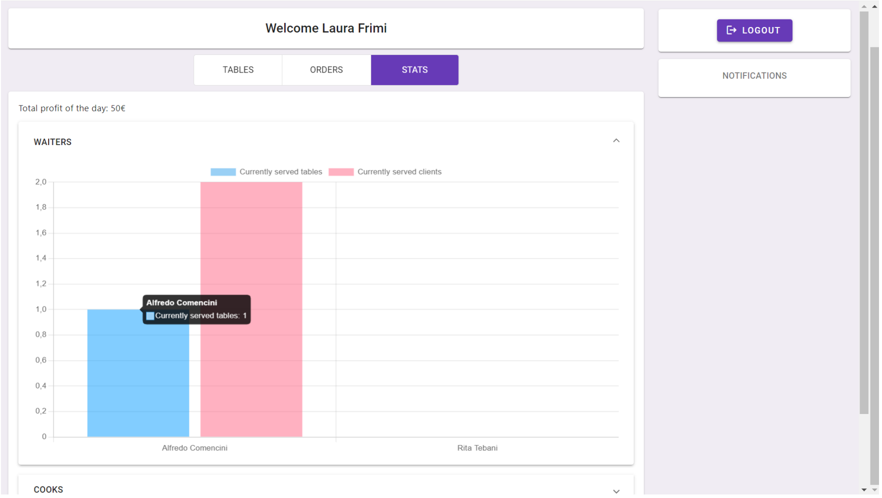
Figure 4.8: The cashier is ready to close the order and has printed the receipt (Desktop)



Welcome Laura Frimi

Total profit of the day: 50€

Figure 4.9: Statistics page (Desktop)



Welcome Laura Frimi

Total profit of the day: 50€

Alfredo Comencini
Currently served tables: 1

Figure 4.10: Analysis of the waiter-related statistics (Desktop)

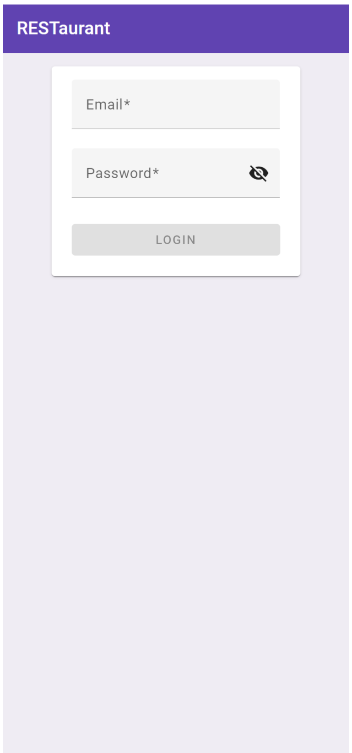
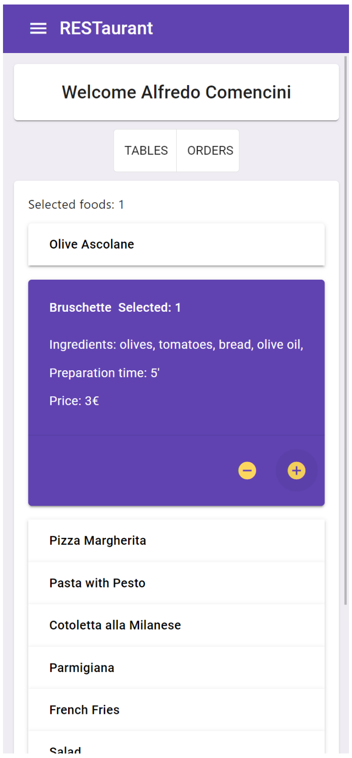


Figure 4.11: Login page (mobile)



Welcome Alfredo Comencini

Selected foods: 1

Olive Ascolane

Bruschette Selected: 1

Ingredients: olives, tomatoes, bread, olive oil,

Preparation time: 5'

Price: 3€

Pizza Margherita

Pasta with Pesto

Cotoletta alla Milanese

Parmigiana

French Fries

Salad

Figure 4.12: Food selection page (mobile)

Chapter 5

Further evolution

Even though the system is complete and functioning by himself, there still is space for some improvements or user experience enhancements.

First of all, in the Table data model could be inserted a field `is_reserved` to distinguish between free tables and tables that have been reserved and are already booked. In that case either it should be impossible to occupy it, or it could be possible only if it would be freed before the reservation time.

Another improvement could be adding notes to the order, for example if a client doesn't want a specific ingredient in a food or if he has some tweaking or adjustment to do. In that case there should be the possibility to add a new note on every order and the cook must be able to access it.

Finally, a rich mechanism to distinguish between different types of food already exists in the Food data model, even though in the app it is barely used, only to distinguish between solid and liquid foods. So a possible improvement could be a more intense utilisation of this field, for example to create some filters or a more organised menu.