

Web Applications and Technologies

Project 2022/23

MEAN Restaurant App

Indice

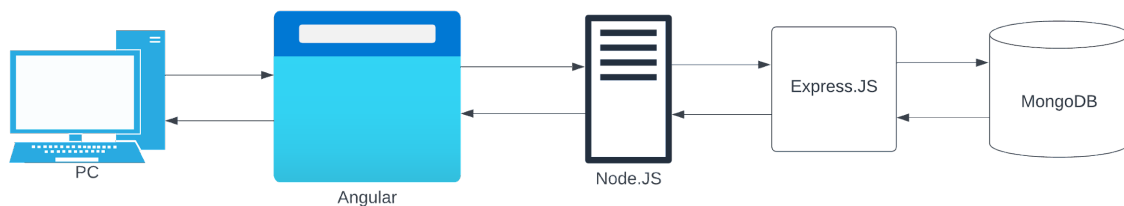
Indice	1
Introduction	2
System Architecture	2
How to Run	2
Data Model	2
Mongoose	5
REST APIs	7
User Authentication	13
Sign Up	16
Angular Frontend	16
Components	17
Services	17
Routes	18
Application Examples	19
Cashier	19
Waiter	23
Cook	26
Bartender	28

Introduction

The project consists in developing a full-stack web application for a restaurant order management system, using the MEAN (MongoDB, Express.js, Angular and Node.js) stack. The users are divided into four different roles (cashier, waiter, cook and bartender), where everyone can access different information and functionalities.

System Architecture

The application is developed using the MEAN stack, so the principal components of the systems architecture are MongoDB, Express.JS, Angular e Node.JS.



The user from localhost can access the Angular application using a web browser. The frontend connects to the server to retrieve all the information he needs to be executing correctly. The server has to query this data from the database (MongoDB) and uses the framework Express.JS to implement the backend API. So for every execution Angular makes a request to the server, that asks through Express.JS the data to MongoDB and then the response follows the opposite path.

How to Run

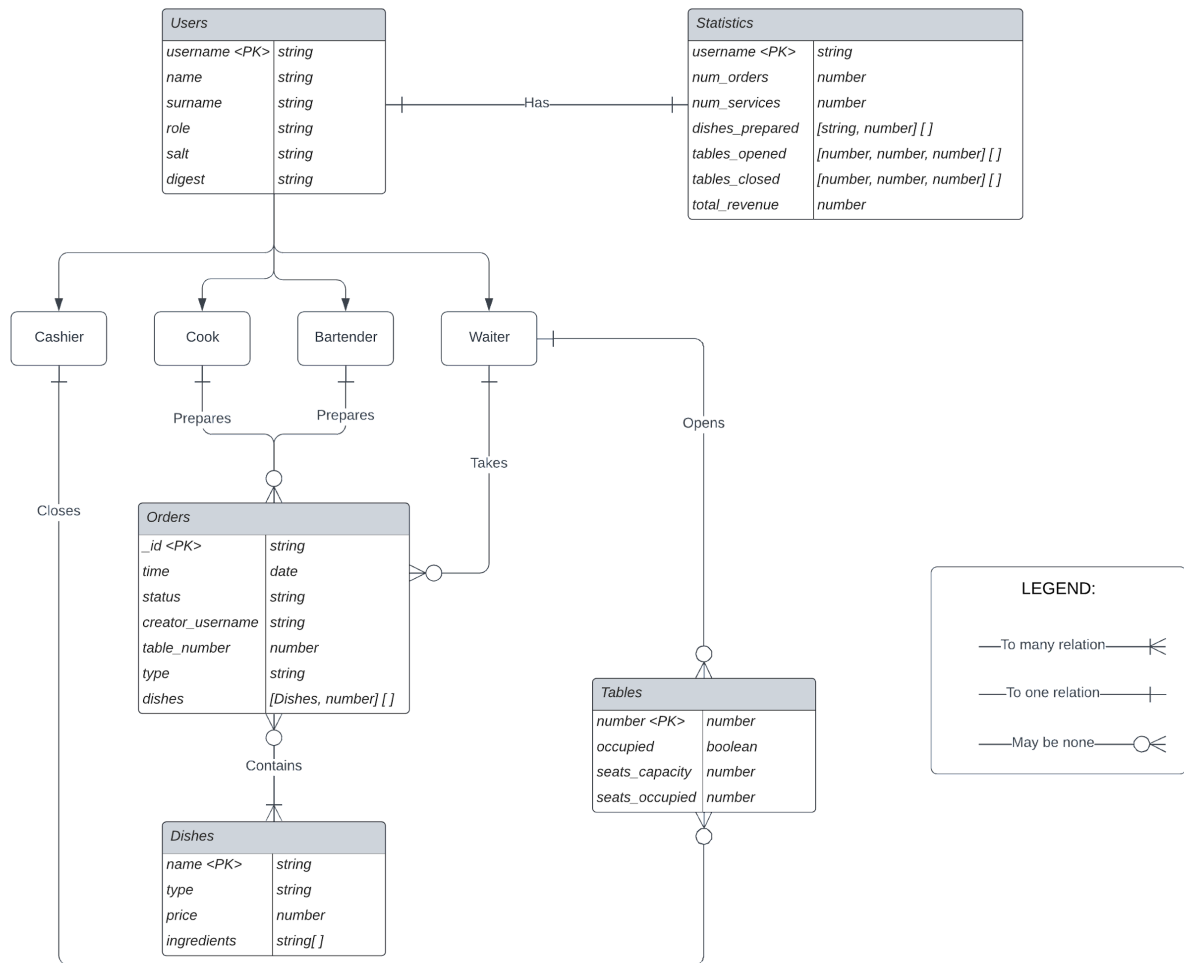
To run the application is needed to follow the instructions written in the README.md file in the principal folder of the project. It contains directives to run the project using Docker, it will create three different containers: one for MongoDB, one for Node.JS and one for Angular.

Data Model

As said before, MongoDB is the database used for storing all the application data. The model is composed of 5 entities: *users*, *tables*, *orders*, *dishes* and *statistics*. Here is inserted the relational schema representing the major relationships between the database entities. Everyone of these tables have different fields to store useful information for the application logic:

- **Users:** this entity refers to the users that use the application and they have a username, which is also the primary key, a name and a surname, a role, that can be 'cashier', 'cook', 'bartender' or 'waiter', a salt and a digest for the operation of identification. Different roles have different privileges and accessible operations, for example a waiter can open tables and take orders, differently a cook or a bartender can prepare orders. The cashier is also an administrator and he can close tables as well as handle the entire organizational system, like adding and deleting tables, dishes and users.

- **Tables:** the table entity has everything useful to know for monitoring the seats during the service. It has a number used as an identifier, a boolean field to check if the table is occupied or not, the capacity of the table and the effective seats occupied if the table is taken.
- **Dishes:** the dishes have a name, unique and used as a primary key, a type that can differ between *'food'* or *'drink'* so it can be handled by cooks or bartenders, a price that represents the cost of the dish and all the ingredients used in the dish preparations.
- **Orders:** this table has an identifier for primary key, a time when the order is taken, a status that represents the current stage in the dish preparation queue (*'todo'*, *'in progress'*, *'to serve'* or *'done'*), the username of the waiter that has taken the order, the number of the table from which people has ordered, the type (*'food'* or *'drink'*) to choose to who send the order and the list of dishes ordered and grouped by name and amount.
- **Statistics:** every user has a related statistics table to store all the useful information about the work that he has done. The entity contains the user's username, the number of orders that he has taken, prepared or closed based on his role, the number of services (workdays) that he has made, the total revenue if he is a cashier and a different list based on the role:
 - *dishes prepared*, if the user is a cook or a bartender, with the format [name of the dish, amount prepared].
 - *tables opened*, if the user is a waiter, with the format [number of the table, how many times the table has been opened, the total number of people served].
 - *tables closed*, if the user is a cashier, with the format [number of the table, how many times the table has been closed, the total amount spent by the table].



Here is an example for each type of collection:

- **User:**

```

{
  username: 'user1',
  name: 'Marco',
  surname: 'Rossi',
  role: 'cashier',
  salt: '7268f3dca84b59affad66457568f97fd',
  digest: '8b02f7058833722022247c4ab29789e...',
};

```

- **Table:**

```

{
  number: 3,
  occupied: true,
  seats_capacity: 7,
  seats_occupied: 6,
}

```

- **Dish:**

```
{
  name: 'Pizza Margherita',
  type: 'food',
  price: 5.5,
  ingredients: [ 'pasta', 'pomodoro', 'mozzarella' ],
}
```

- **Order:**

```
{
  _id: ObjectId("6491d44d6e800c6a632185e4"),
  time: ISODate("2023-06-20T16:31:09.008Z"),
  status: 'todo',
  creator_username: 'user1',
  table_number: 5,
  type: 'drink',
  dishes: [
    [
      {
        name: 'Coca Cola',
        type: 'drink',
        price: 2.5,
        ingredients: []
      },
      1
    ]
  ]
}
```

- **Statistic:**

```
{
  username: 'user2',
  num_orders: 4,
  num_services: 4,
  dishes_prepared: [ [ 'Spaghetti allo Scoglio', '2' ], [
    'Pizza Margherita', '2' ] ],
}
```

Mongoose

The project uses the library Mongoose to handle MongoDB objects through TypeScript objects and to have an easier way to query the database. Here is an example of a data model (User), the other models are similar, the only differences are the fields defined.

```
import mongoose = require('mongoose');
import crypto = require('crypto');

export interface User extends mongoose.Document {
  username: string,
  name: string,
  surname: string,
  role: string,
  salt: string,
  digest: string,
  setPassword: (password: string) => void,
  checkPassword: (password: string) => boolean,
  isAdmin: () => boolean,
  setAdmin: () => void
}

const ADMIN: string = "cashier";
const ROLES: string[] = ["cashier", "waiter", "bartender", "cook"];

const userSchema = new mongoose.Schema<User>({
  username: {
    type: mongoose.SchemaTypes.String,
    required: true,
    unique: true
  },
  name: {
    type: mongoose.SchemaTypes.String,
    required: true
  },
  surname: {
    type: mongoose.SchemaTypes.String,
    required: true
  },
  role: {
    type: mongoose.SchemaTypes.String,
    enum: ["cashier", "waiter", "bartender", "cook"],
    required: true
  },
  salt: {
    type: mongoose.SchemaTypes.String,
    required: false
  },
  digest: {
    type: mongoose.SchemaTypes.String,
    required: false
  }
});
```

```

digest: {
  type: mongoose.SchemaTypes.String,
  required: false
}
}))
. . .

```

REST APIs

The backend of the web app consists in a Node.js server implementing a list of REST APIs to access the data model and compute some operations. The file index in the backend folder contains a list of Express.js middlewares and routing rules, which are specified below:

- **/users:**
 - *GET /users:* route that returns the list of all the users registered in the application

```

[
  {
    "_id": "6491d9f855c81e4d1f92e4b7",
    "username": "admin",
    "name": "admin",
    "surname": "admin",
    "role": "cashier",
    "__v": 0
  },
  {
    "_id": "6491d9f955c81e4d1f92e4d6",
    "username": "user2",
    "name": "Mario",
    "surname": "Verdi",
    "role": "cook",
    "__v": 0
  },
  . . .
]

```

- *POST /users:* route to add a new user with the data sent through the body

```

{
  "username": "new_user",
  "password": "new_pwd",
  "name": "Mattia",
  "surname": "Verdi",
  "role": "cook"
}

```

- **/users/:username:**

- *GET /users/:username:* route that retrieves all the data of the user specified
- *DELETE /users/:username:* route that deletes the user specified

```
{
  "error": false,
  "errorMessage": ""
}
```

- *PUT /users/:username:* route to update the user data with information sent through the body

```
{
  "name": "Alice"
}
```

```
{
  "acknowledged": true,
  "modifiedCount": 1,
  "upsertedId": null,
  "upsertedCount": 0,
  "matchedCount": 1
}
```

- **/dishes:**

- *GET /dishes:* route that returns the list of all the dishes presents in the restaurant menu
- *POST /dishes:* route to add a new dish with the data sent through the body

```
{
  "name": "Hamburger",
  "type": "food",
  "price": 9.5,
  "ingredients": ["Meat", "Cheese", "Onion", "Tomato", "Bread"]
}
```

- **/dishes/:name:**

- *GET /dishes/:name:* route that retrieves all the data of the dish specified

```
{
  "_id": "6491d9f955c81e4d1f92e4bf",
  "name": "Coca Cola",
  "type": "drink",
  "price": 2.5,
  "ingredients": [],
  "__v": 0
}
```


- *DELETE /dishes/:name*: route that deletes the dish specified
- **/tables:**
 - *GET /tables*: route that returns the list of all the restaurant's tables

```
[
  {
    "_id": "6491d9f955c81e4d1f92e4ca",
    "number": 4,
    "occupied": false,
    "seats_capacity": 4,
    "seats_occupied": 0,
    "__v": 0
  },
  {
    "_id": "6491d9f955c81e4d1f92e4c9",
    "number": 3,
    "occupied": true,
    "seats_capacity": 7,
    "seats_occupied": 6,
    "__v": 0
  }
  ...
]
```

- *POST /tables*: route to add a new table with the data sent through the body

```
{
  "number": 10,
  "seats_capacity": 5
}
```

- **/tables/:number:**

- *GET /tables/:number*: route that retrieves all the data of the table specified
- *DELETE /tables/:number*: route that deletes the table specified
- *PUT /tables/:number*: route to update the table data with information sent through the body

```
{
  "occupied": true,
  "seats_occupied": 2
}
```

```
{
  "acknowledged": true,
  "modifiedCount": 1,
  "upsertedId": null,
}
```

```
"upsertedCount": 0,  
"matchedCount": 1  
}
```

- **/orders:**

- *GET /orders*: route that returns the list of all the orders taken
- *POST /orders*: route to add a new order with the data sent through the body

```
{  
  "creator_username": "user4",  
  "table_number": 1,  
  "type": "drink",  
  "dishes": [  
    [  
      {  
        "name": "Coca Cola",  
        "type": "food",  
        "price": 2.5,  
        "ingredients": []  
      },  
      3  
    ]  
  ]  
}
```

- **/orders/:id:**

- *GET /orders/:id*: route that retrieves all the data of the order specified

```
{  
  "_id": "6491d9fa55c81e4d1f92e4e2",  
  "time": "2023-06-20T16:55:19.668Z",  
  "status": "todo",  
  "creator_username": "user1",  
  "table_number": 5,  
  "type": "food",  
  "dishes": [  
    [  
      {  
        "name": "Pizza Margherita",  
        "type": "food",  
        "price": 5.5,  
        "ingredients": [  
          "pasta",  
          "pomodoro",  
          "mozzarella"  
        ]  
      }  
    ]  
  ]  
}
```

```

    },
    2
  ]
],
"__v": 0
}

```

- *DELETE /orders/:id*: route that deletes the order specified
- *PUT /orders/:id*: route to update the order data with information sent through the body

```

{
  "status": "in progress"
}

```

- **/statistics:**

- *GET /statistics*: route that returns the list of all the statistics
- *PUT /statistics*: route that updates all the statistics

```

{
  "num_orders": 0
}

```

```

{
  "acknowledged": true,
  "modifiedCount": 4,
  "upsertedId": null,
  "upsertedCount": 0,
  "matchedCount": 6
}

```

- **/statistics/:username:**

- *GET /statistics/:username*: route that retrieves all the data of the statistic specified

```

{
  "_id": "6491d9fa55c81e4d1f92e4ec",
  "username": "user4",
  "num_orders": 4,
  "num_services": 2,
  "dishes_prepared": [],
  "tables_opened": [
    4,
    2,
    3
  ],
}

```

```
[
  3,
  1,
  6
],
[
  2,
  1,
  2
]
],
"tables_closed": [],
"total_revenue": 0,
"__v": 0
}
```

- ```
{
 "num_orders": 5
}
```

```
{
 "acknowledged": true,
 "modifiedCount": 1,
 "upsertedId": null,
 "upsertedCount": 0,
 "matchedCount": 1
}
```

- **/login:**

- ```
{  
  "error": false,  
  "errorMessage": "",  
  "token":  
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluliwibmFtZSI6ImFkbWluliwic3VybmFtZSI6ImFkbWluliwicm9sZSI6ImNhc2hpZXIiLCJpYXQiOjE2ODcyODAyMzQslmV4cCI6MTY4NzMwMTgzNH0.4-zEh5lF0kSp7LFUnXKxiWHKhgmcWUkmrLuKSyVI3o"  
}
```

Every route, except for the login route, requires passing the jwt authentication middleware and some routes, like the post methods, have to pass a middleware that checks that the user has an admin role (cashier).

There are also some error handling middlewares that are executed if there are errors in the requests or if an invalid endpoint is inserted.

User Authentication

When the user opens the application for the first time it will be redirected to the login page that is inserted below.



A login form with a cloud icon at the top. Below the icon is the text "Hi, sign in!". There are two input fields: "Username" and "Password". Below the "Password" field is a blue "Sign in" button. At the bottom of the form is a copyright notice: "© 2017-2023".

The page contains a form to insert the username and the password and if they are correct the user will be redirected to the dashboard to use the application functionalities. When the sign in button is clicked it is called a method of an Angular service, called login(username, password) in the user-http.service.ts file:

```
login(username: string, password: string): Observable<any> {
  const options = {
    headers: new HttpHeaders({
      authorization: 'Basic ' + btoa(`${username}:${password}`),
      'cache-control': 'no-cache',
      'Content-Type': 'application/x-www-form-urlencoded',
    })
  };

  return this.http.get(this.url+"/login", options).pipe(
    tap((data) => {
      console.log(JSON.stringify(data));
      this.token = (data as ReceivedToken).token;
      localStorage.setItem("restaurant_app_token", this.token as
string);
    })
  );
}
```

This function calls the get HTTP method at the route /login, defined in the backend of the application, and then sets the authentication token received as response in the browser, so the user can access the application without the login form next time.

In the backend the login is handled using the passport JS middleware with Basic Authentication. It checks whether the username exists and the password for the user is correct, if it is, the authentication token is given back from the GET method.

```
passport.use(new passportHTTP.BasicStrategy(
  function(username, password, done) {
    user.getModel().findOne({username: username}, {}).then((user)
=> {
      if (!user)
        return done(null, false, {statusCode: 500, error: true,
errormessage: "Invalid user"});

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

      return done(null, false, {statusCode: 500, error: true,
errormessage: "Invalid password"});
    }).catch((err) => {
      return done({statusCode: 500, error: true, errormessage:
err});
    });
  }
));

app.get("/login", passport.authenticate("basic", {session: false}),
(req, res, next) => {
  let tokendata = {
    username: req.user.username,
    name: req.user.name,
    surname: req.user.surname,
    role: req.user.role
  };

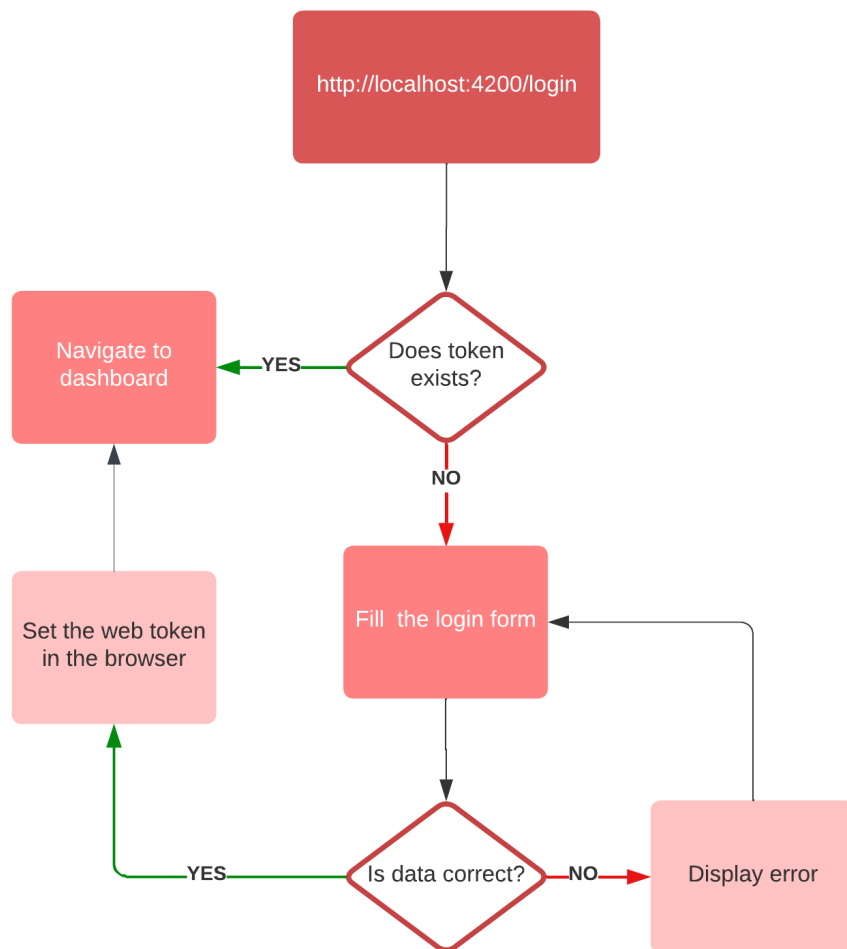
  console.log("Login granted. Generating token" );
  let token_signed = jsonwebtoken.sign(tokendata,
process.env.JWT_SECRET, { expiresIn: '6h' } );

  return res.status(200).json({ error: false, errormessage: "",
token: token_signed });
});
```

Everytime the user is redirected to the login page in the frontend, the web token is checked in the Angular login component and if it exists and it is valid the user is automatically sent to the dashboard, otherwise he has to fill the access form.

```
ngOnInit(): void {  
    if (!this.jhs.isTokenExpired(this.us.get_token()))  
  
this.router.navigate([this.us.dashboard_routes[this.us.get_role()]]);  
}
```

Summing up the login logic can be represented with the following diagram:



The user has access to any application route only after the mandatory login, to ensure this it has been implemented a guard service that check the token before access any route, if it is not valid the user is redirected to the login page.

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):  
Observable<boolean> | boolean {  
    if (state.url !== '/login' &&  
this.jhs.isTokenExpired(this.us.get_token())) {
```

```

        this.router.navigate(['login']);
        return false;
    }

    return true;
}

```

Sign Up

The sign up operation is the responsibility of the cashier users, they can see the complete list of the staff, delete some members and add new ones. They insert every information of the new users, like username, name, surname, role, and also the password. After the first login the new staff member can change his password as he wants in his profile section of the application.

Angular Frontend

The Angular application contains all the logic needed to run the frontend of the restaurant app. Here is inserted the app.module.ts file containing the list of all the components and all the services implemented in the /app folder.

```

@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
    TablesComponent,
    NavbarComponent,
    StaffComponent,
    OrdersComponent,
    MenuComponent,
    StatisticsComponent,
    PageNotFoundComponent,
    ProfileComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [
    {provide: UserHttpService, useClass: UserHttpService },
    {provide: TableHttpService, useClass: TableHttpService },
    {provide: OrdersHttpService, useClass: OrdersHttpService },

```



```

    {provide: DishHttpService, useClass: DishHttpService },
    {provide: StatisticsHttpService, useClass: StatisticsHttpService },
    {provide: SocketioService, useClass: SocketioService },
    {provide: GuardService, useClass: GuardService },
    {provide: JwtHelperService, useClass: JwtHelperService },
    { provide: JWT_OPTIONS, useValue: JWT_OPTIONS }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Components

The components are grouped by the data they have to display or handle, everyone of them changes based on the user role or on the current application route. Here are listed all the components followed by a brief description:

- **LoginComponent:** contains the form code and the login method previously seen.
- **TablesComponent:** contains all the logic useful for the list of the tables visualization and the modification of them.
- **NavbarComponent:** is used to implement the navbar that the users used to navigate inside the application and it changes based on the user role.
- **StaffComponent:** presents all the code used to handle the staff list and his modification, like adding or removing users.
- **OrdersComponent:** contains all the logic used to display the list of orders, giving the possibility to modify them and compute useful information.
- **MenuComponent:** is used to visualize the list of dishes with their information, adding them and removing them.
- **StatisticsComponent:** contains all the logic to compute and display the users statistics.
- **PageNotFoundComponent:** presents the code to handle the case in which the user navigates to a not existing route.
- **ProfileComponent:** is used to display the user data and to change the user password.

Services

Services present all the functions needed to the components to be executed successfully and they are:

- **UserHttpService:** implements the methods used to make APIs calls refers to the user logic.
- **TableHttpService:** implements the methods used to make APIs calls refers to the table logic.
- **OrdersHttpService:** implements the methods used to make APIs calls refers to the order logic.
- **DishHttpService:** implements the methods used to make APIs calls refers to the dish logic.

- **StatisticsHttpService:** implements the methods used to make APIs calls refers to the statistics logic.
- **SocketioService:** implements the methods connect and get_update to implement the possibility to see changes in real time in some components.
- **GuardService:** implements the method canActivate to check if a user is entering a route without being logging in or is entering a non-existent route.

Routes

Here is possible to visualize the different routes with the associated components:

```
const routes: Routes = [
  {path: "", redirectTo: "/login", pathMatch: "full"},
  {path: "login", component: LoginComponent},
  {path: "tables", component: TablesComponent, canActivate:
[GuardService]},
  {path: "staff", component: StaffComponent, canActivate:
[GuardService]},
  {path: "menu", component: MenuComponent, canActivate:
[GuardService]},
  {path: "orders", component: OrdersComponent, canActivate:
[GuardService]},
  {path: "orders/table/:number", component: OrdersComponent,
canActivate: [GuardService]},
  {path: "order/table/:number", component: MenuComponent, canActivate:
[GuardService]},
  {path: "statistics/:username", component: StatisticsComponent,
canActivate: [GuardService]},
  {path: "statistics", component: StatisticsComponent, canActivate:
[GuardService]},
  {path: "profile", component: ProfileComponent, canActivate:
[GuardService]},
  {path: '**', component: PageNotFoundComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

The login route is related to the access process, the staff, menu, orders and statistics routes display a general list of their data information, the profile route displays the user personal information, the ** is dedicated to all the non-existent routes and the remaining ones are dedicated to one specific element of their data lists.

Application Examples

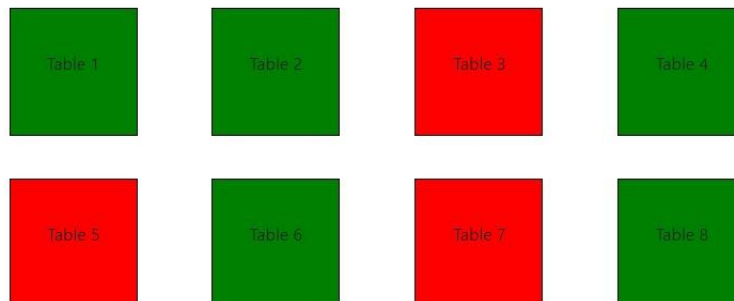
This section contains some examples of the application execution, divided by the user's role.

Cashier

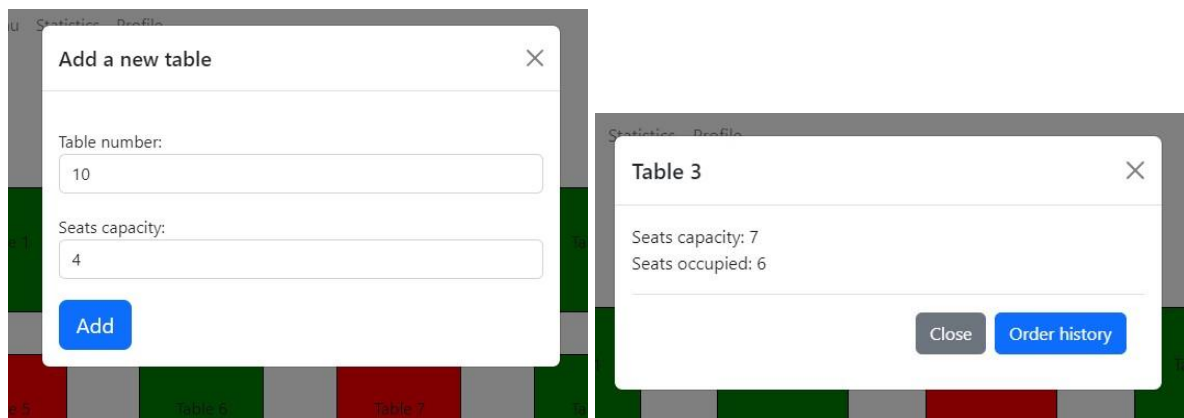
After the login, the user navigates to his dashboard where he can see the list of all the tables, green if the table is free and red otherwise. If he clicks on a free table he can decide to delete it, instead on an occupied table he can see the table's orders history. Under the table list there is a button that opens a form to add a new table.

 [Tables](#) [Orders](#) [Staff](#) [Menu](#) [Statistics](#) [Profile](#) [Logout](#) 

Tables



Add table



In the orders history is present the list of all the orders taken for the table specified and the cashier can produce the final receipt and compute the payment operation through the pay button in the bottom of the page.



Table 3 Orders History

Food:

10:04 AM, *table 3*

- penne pomodoro x1
- pizza margherita x1

Creator: *user6*

todo

Drinks:

10:04 AM, *table 3*

- coca cola x2

Creator: *user6*

todo

People: 6

Total price: 31.5€

Payout

Confirmation

×

coca cola x2: 6€
penne pomodoro x1: 7€
pizza margherita x1: 6.5€
Fee x6: 12€

Are you sure to pay 31.5€?

Close

Pay

Clicking on orders in the navbar the user can see the list of all the orders for the different tables with their current status.



Orders List

Food:

10:04 AM, *table 3*

- penne pomodoro x1
- pizza margherita x1

Creator: *user6*

todo

10:04 AM, *table 7*

- beef x1
- mixed fried x1

Creator: *user8*

todo

10:04 AM, *table 5*

- pizza margherita x2

Creator: *user7*

todo

Drinks:

10:04 AM, *table 3*

- coca cola x2

Creator: *user6*

todo

10:04 AM, *table 5*

- coca cola x1

Creator: *user7*

todo










The staff section contains the list of all the staff members with the possibility of removing someone or to add a new user, using the dedicated button. Like for every form if there are problems adding something an error message is displayed in the modal opened. Clicking on the name of a user is possible to see his statistics, it will be shown further on the report.



[Tables](#) [Orders](#) [Staff](#) [Menu](#) [Statistics](#) [Profile](#)

[Logout](#)

Staff

user1 Marco Rossi	cashier 
user2 Mario Verdi	cook 
user3 Gianluca Zorti	cook 
user4 Alice Barro	bartender 
user5 Davide Mattei	bartender 
user6 Giovanni Gioia	waiter 
user7 Anna Remi	waiter 
user8 Annalisa Pozzi	waiter 
user9 Marco Fausti	waiter 

Add member

Confirmation

Are you sure you want to delete user5?

Close

Delete

Add a staff member

Username

user1

Password

.....

Name

Matteo

Surname

Bianchi

Role

Cashier

Adding error: User user1 already exists.

Add

The menu section is similar to the staff one, in fact it displays the list of all the dishes in the menu, divided by their type, with the possibility of removing a dish or adding a new one.



[Tables](#) [Orders](#) [Staff](#) [Menu](#) [Statistics](#) [Profile](#)

[Logout](#)

Menu

Food:

beef Ingredients: beef, rocket, cherry tomatoes	10€
pizza margherita Ingredients: pasta, pomodoro, mozzarella	6.5€
penne pomodoro Ingredients: pasta, tomato, sea fruits	7€
mixed fried Ingredients: shrimp, squid, sardines	10.5€

Drinks:

medium water Ingredients:	2.5€
coca cola Ingredients: carbonated water, sugar	3€
medium bier Ingredients: grain, hops, yeast, water	4.5€

Add dish

The statistics page shows the general statistics of the restaurant, like the number of staff members, the total revenue and other daily data. The statistics related to the day can be removed clicking on the reset statistics button.

As said before, it is possible to see the statistics for a specific user clicking on his username in the staff members list.



[Tables](#) [Orders](#) [Staff](#) [Menu](#) [Statistics](#) [Profile](#)

[Logout](#)

Statistics



Number of staff members:

10

Total revenue:

300€

Daily number of tables served:

12

Daily number of dishes prepared:

13

Daily number of people served:

41

Reset statistics



user1 statistics



Name and Surname:

Marco Rossi

Role:

cashier

Number of Services:

4

Daily number of Orders Checked:

26

Daily total revenue:

185€

Daily tables_closed:

Table number 1 x2, Total amount: 120€

Table number 3 x1, Total amount: 40€

Table number 6 x1, Total amount: 25€

Every user has also a profile section where he can see his personal information and change his current password.



Profile



Username:

admin

Name:

admin

Surname:

admin

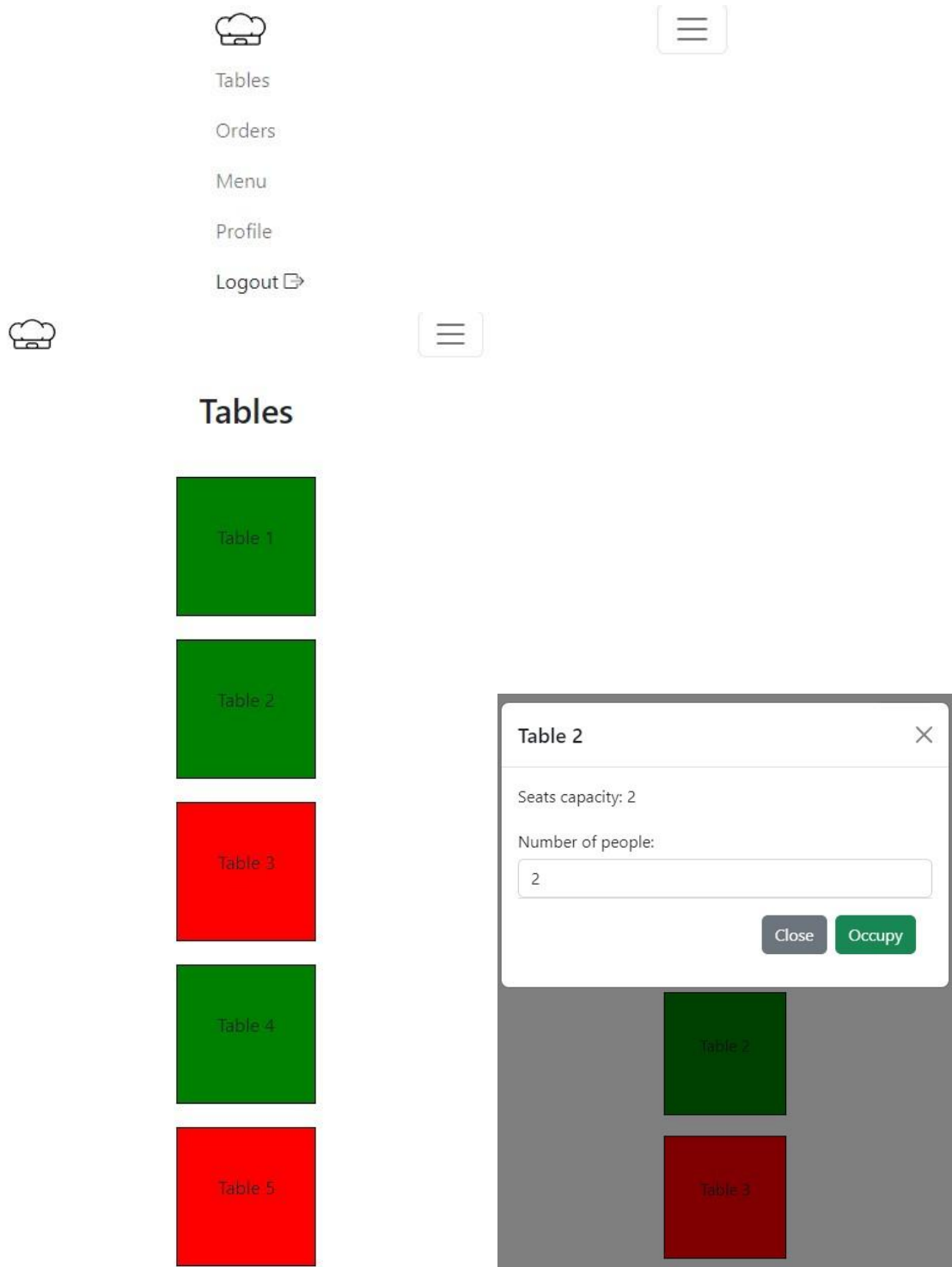
Role:

cashier

[Change Password](#)

Waiter

The waiter's dashboard is always the list of the restaurant's tables, but the interface is adapted to a smartphone screen. A user clicking on a free table can occupy the table inserting the number of people, that must be less or equal than the seats capacity. Clicking on an occupied table instead is possible to see its order history.



The table's order history consists in a list of all the orders of the table and the waiter can delete some orders, clicking on the delete option on the right of each item. Clicking on the order button it is possible to create a new order. It will open a menu list where a user can choose the amount for each dish and then confirm the order.



Table 3 Orders History

Food:

10:04 AM, <i>table 3</i>	Creator: <i>user6</i> <i>Delete</i>
◦ penne pomodoro x1	todo
◦ pizza margherita x1	
10:04 AM, <i>table 3</i>	Creator: <i>user7</i> <i>Delete</i>
◦ beef x2	todo

Drinks:

10:04 AM, <i>table 3</i>	Creator: <i>user6</i> <i>Delete</i>
◦ coca cola x2	todo

People: 6

Order

beef

Amount: 0

Ingredients: beef, rocket, cherry tomatoes

+
—

pizza margherita

Amount: 2

Ingredients: pasta, pomodoro, mozzarella

+
—

penne pomodoro

Amount: 0

Ingredients: pasta, tomato, sea fruits

+
—

mixed fried

Amount: 1

Ingredients: shrimp, squid, sardines

+
—

Drinks:

medium water

Amount: 0

Ingredients:

+ —

coca cola

Amount: 1

Ingredients: carbonated water, sugar

+
—

medium bier

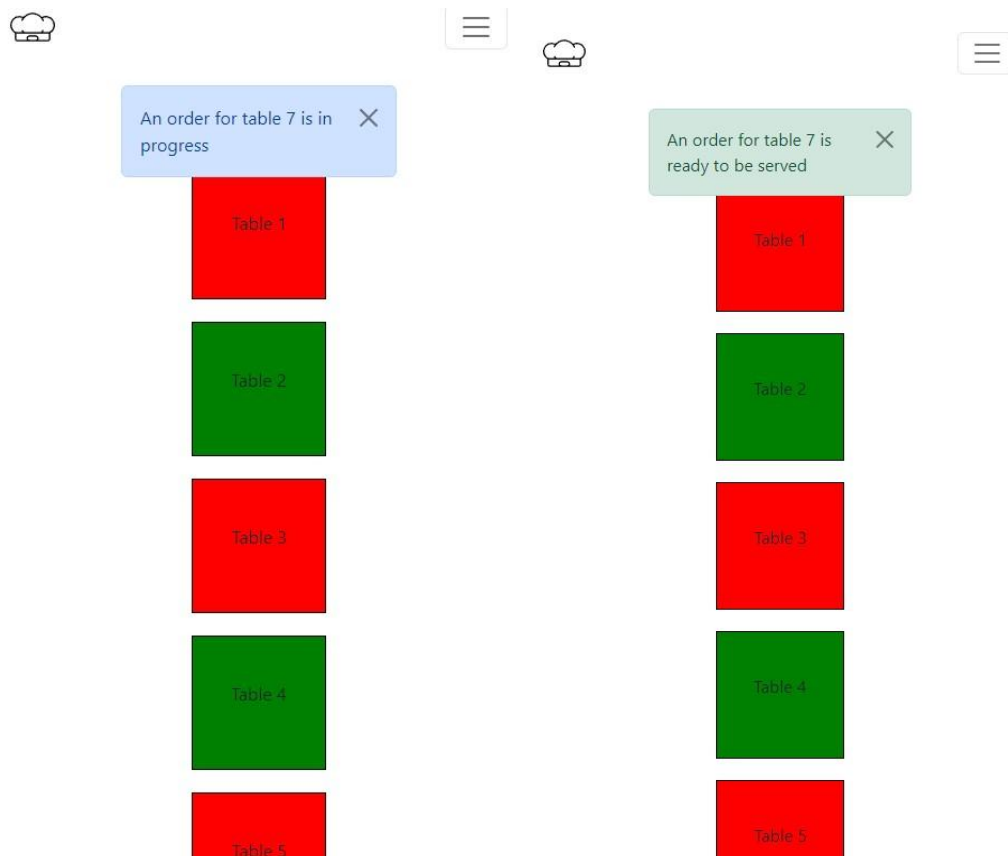
Amount: 1

Ingredients: grain, hops, yeast, water

+
—

Make order

When a cook or a bartender changes the status of an order in *'in progress'* or in *'to serve'*, a waiter is notified with the updating status message.



The navbar sections for the menu, for all the orders and for the profile simply shows data that has been inserted in the waiter part.

Cook

The cook's dashboard consists in the list of all the orders related to the food dishes. A user can see the current order's status and, clicking on it, he can change it. The list is ordered with the FIFO logic, but orders with a better status are placed before the others. The status order is the following one, from the highest in the list to the lowest: to serve, in progress, to do, done.

Orders List

Food:

10:04 AM, table 3 <ul style="list-style-type: none"> ◦ penne pomodoro x1 ◦ pizza margherita x1 	Creator: user6 todo
10:04 AM, table 7 <ul style="list-style-type: none"> ◦ beef x1 ◦ mixed fried x1 	Creator: user8 todo
10:04 AM, table 5 <ul style="list-style-type: none"> ◦ pizza margherita x2 	Creator: user7 todo

Update status

Status

In Progress

Change



Orders Menu Profile

Logout

Orders List

Food:

10:04 AM, <i>table 7</i> <ul style="list-style-type: none">beef x1mixed fried x1	Creator: <i>user8</i> in progress
10:04 AM, <i>table 3</i> <ul style="list-style-type: none">penne pomodoro x1pizza margherita x1	Creator: <i>user6</i> todo
10:04 AM, <i>table 5</i> <ul style="list-style-type: none">pizza margherita x2	Creator: <i>user7</i> todo

When a waiter takes a new order, it will be added to the dedicated list and the cook receives a notification with the message of the list updating.



Orders Menu Profile

Logout

There is a new order for table 3

Food:

10:04 AM, <i>table 7</i> <ul style="list-style-type: none">beef x1mixed fried x1	Creator: <i>user8</i> in progress
10:04 AM, <i>table 3</i> <ul style="list-style-type: none">penne pomodoro x1pizza margherita x1	Creator: <i>user6</i> todo
10:04 AM, <i>table 5</i> <ul style="list-style-type: none">pizza margherita x2	Creator: <i>user7</i> todo
10:04 AM, <i>table 3</i> <ul style="list-style-type: none">beef x2	Creator: <i>user7</i> todo

Bartender

The bartender interface is the same as the cook's one, with the only exception that the list of orders displays all the data related to the dishes that have drink as a type.

[Orders](#) [Menu](#) [Profile](#)[Logout](#)

Orders List

Drinks:

10:04 AM, <i>table 3</i> <ul style="list-style-type: none">coca cola x2	Creator: <i>user6</i> todo
10:04 AM, <i>table 5</i> <ul style="list-style-type: none">coca cola x1	Creator: <i>user7</i> todo
10:04 AM, <i>table 7</i> <ul style="list-style-type: none">medium water x2	Creator: <i>user8</i> todo