Web Applications and Technologies Project 2022/2023

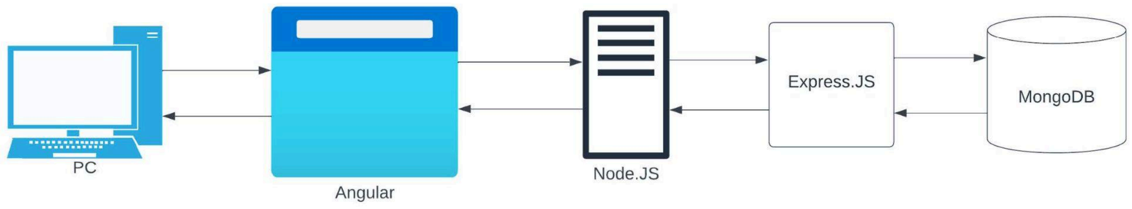
MEAN RESTAURANT APP

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 and 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, which asks through Express.js the data to MongoDB.

How to run

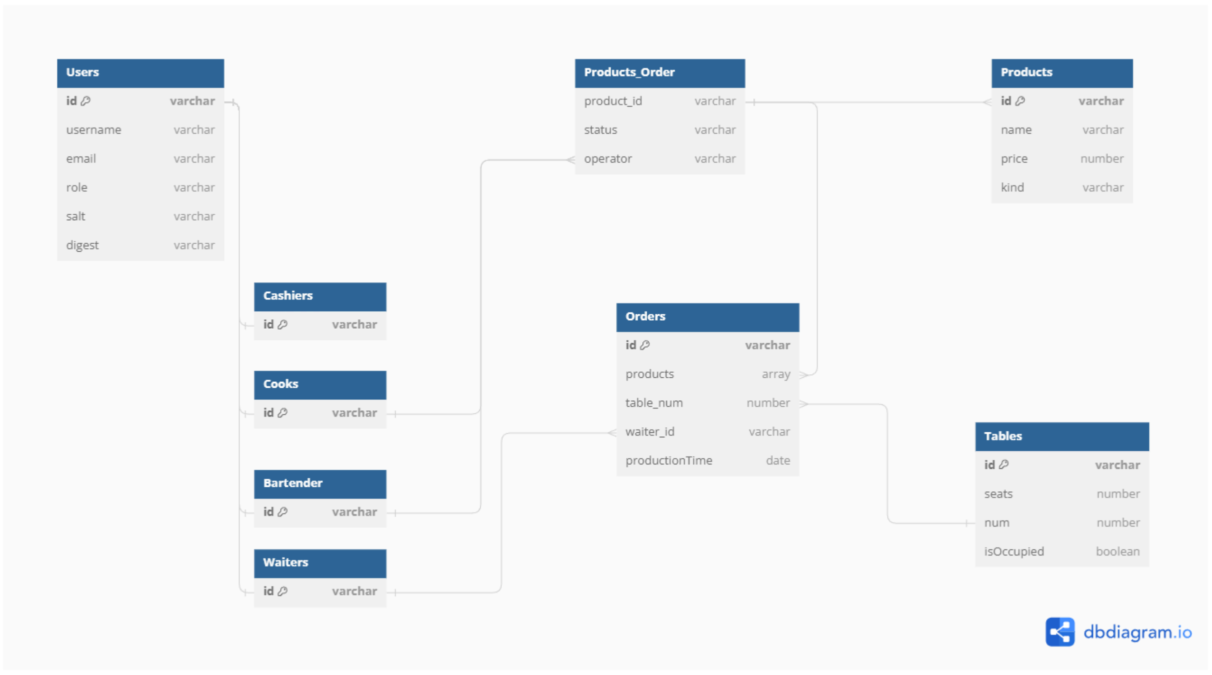
To run the application is needed to follow the instructions written in the README.md file.

Data Model

MongoDB is the database used for storing all the application data. The model is composed of entities: *users, tables, orders, products and statistics*.

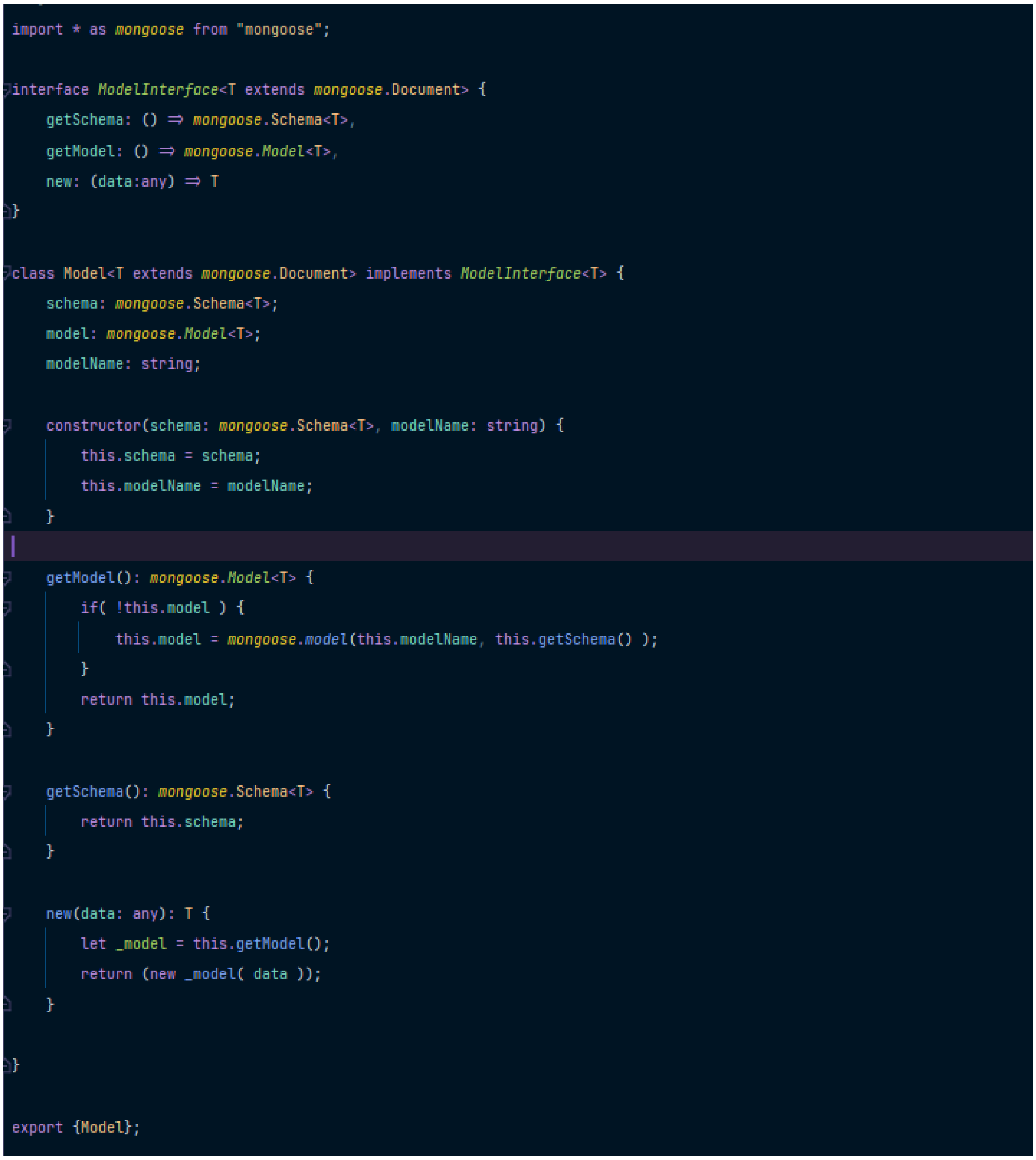
- **Users**: this entity refers to the users that use the application. They have a username that identifies them, an email, a role, a salt and a digest for the identification process. Roles have different functionalities and accessible information. The cashier is also an admin.
- **Tables**: this entity refers to the tables of the restaurant. They have a number that identifies them, a number of seats, a boolean field that checks if the table is occupied.
- **Products**: this entity refers to the products of the menu of the restaurant. They have a unique name, a number that is the price and a string that is the kind of the products (“Food” or “Drink”).

- **Orders**: this entity is composed of: productionTime, which is the time when the order is created, the number of the table, the id of the waiter and an array of *extended products*. An extended product is a product defined as before and some other fields: status, which is a string value of the process status, an id to identify that product in the order.
- **Statistics**: this entity refers to the user and how many services it made. This entity does not include all of the statistics of the application.



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 the template data model on which the models are based.



```typescript
import * as mongoose from "mongoose";

interface ModelInterface<T extends mongoose.Document> {
    getSchema: () => mongoose.Schema<T>,
    getModel: () => mongoose.Model<T>,
    new: (data:any) => T
}

class Model<T extends mongoose.Document> implements ModelInterface<T> {
    schema: mongoose.Schema<T>;
    model: mongoose.Model<T>;
    modelName: string;

    constructor(schema: mongoose.Schema<T>, modelName: string) {
        this.schema = schema;
        this.modelName = modelName;
    }

    getModel(): mongoose.Model<T> {
        if( !this.model ) {
            this.model = mongoose.model(this.modelName, this.getSchema() );
        }
        return this.model;
    }

    getSchema(): mongoose.Schema<T> {
        return this.schema;
    }

    new(data: any): T {
        let _model = this.getModel();
        return (new _model( data ));
    }

}

export {Model};
```

REST API

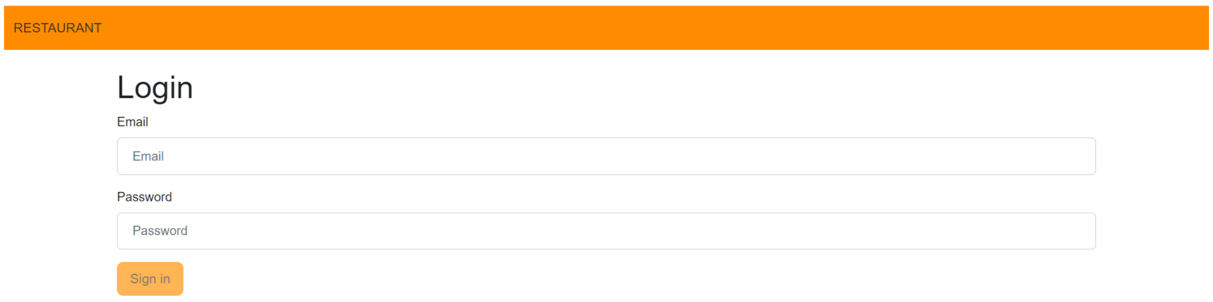
The backend of the web app consists in a Node.js server implementing a list of REST APIs to access the data model and compute some operations.

Endpoints	**Attributes**	**Method**	**Description**
/	-	GET	Returns the version and a list of available endpoints
/login	-	GET	Login an existing user, returning a JWT
/users	-	GET	Returns all the users
/users	-	POST	Add new user
/users/:email	-	GET	Get user info by email
/users/:email	-	PUT	Update user info by email
/users/:email	-	DELETE	Delete user by email
/tables	?isOccupied=	GET	Returns all the tables, optionally filtered
/tables	-	POST	Add new table
/tables/:num	-	PUT	Update table info by num
/tables/:num	-	DELETE	Delete table by num
/tables/:num/orders	-	GET	Returns all orders of a table
/products	-	GET	Returns all the products
/products	-	POST	Add new product
/products/:name	-	GET	Get product info by name
/products/:name	-	PUT	Update product info by name
/products/:name	-	DELETE	Delete product by name
/orders	-	GET	Returns all the orders
/orders	-	POST	Add new order
/orders/:id	-	GET	Get order info by id
/orders/:id	-	PUT	Update order info by id
/orders/:id	-	DELETE	Delete order by id
/orders/:id/products/:product	-	PUT	Update a product of an order by id
/orders/:id/products/:product	-	DELETE	Delete a product of an order by id
/statistics/users	-	GET	Returns all the statistics about users
/statistics/orders	-	GET	Returns all the statistics about orders
/statistics/tables	-	GET	Returns all the statistics about tables

Every route, except for the login and home route, requires authentication.
The routes are accessible based on the role of the user and this is checked with some middlewares. 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.



RESTAURANT

Login

Email

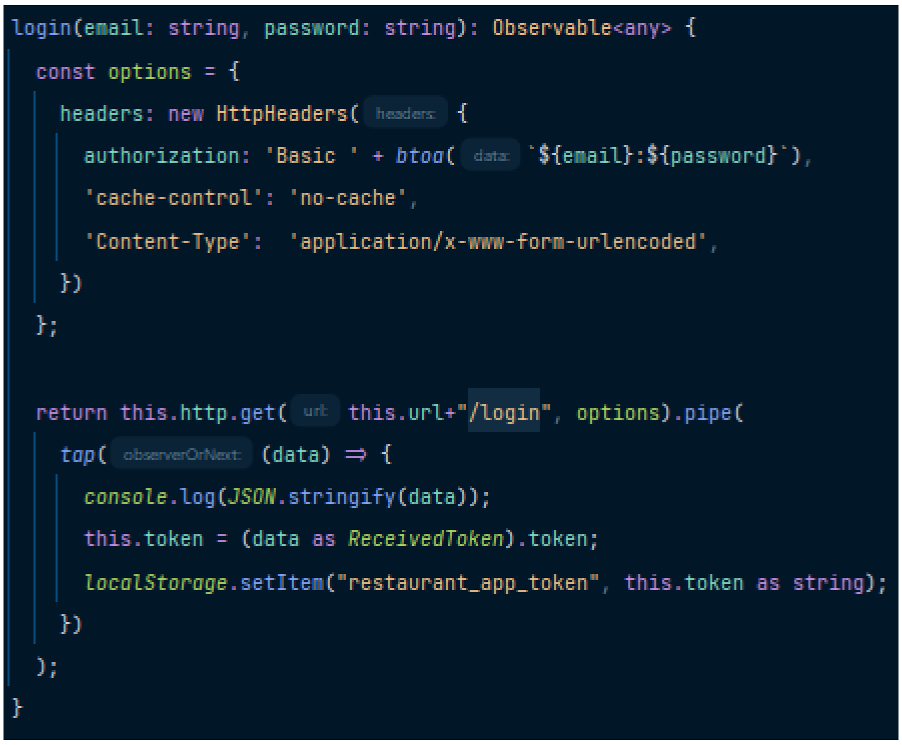
Email

Password

Password

Sign in

The page contains a form to insert the email 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(email, password) in the user.service.ts file.

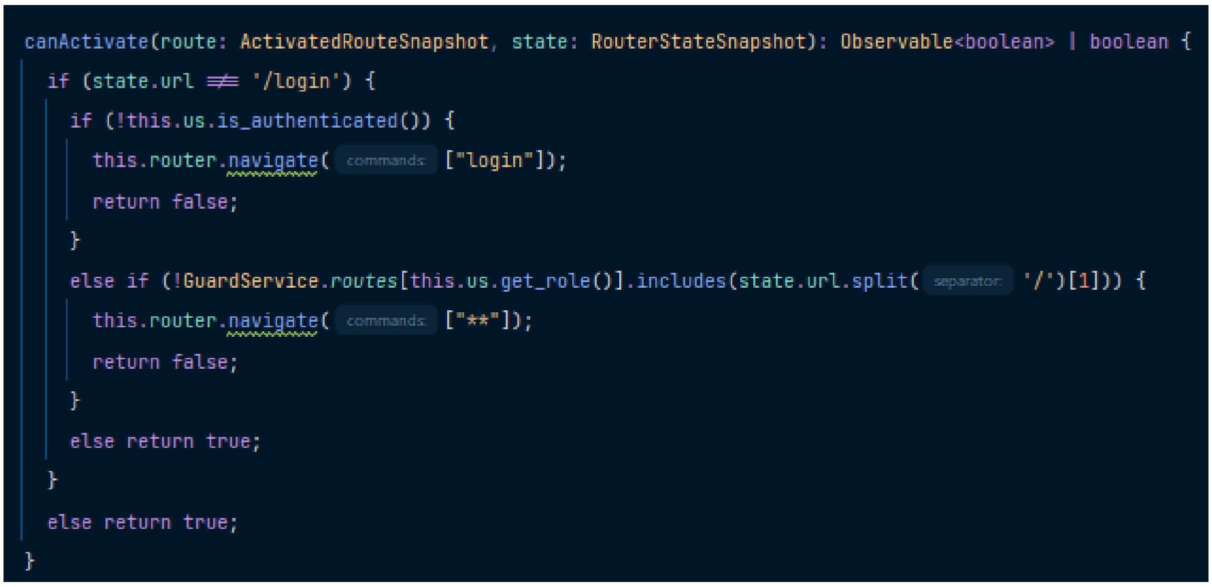


```
login(email: string, password: string): Observable<any> {
  const options = {
    headers: new HttpHeaders( headers: {
      authorization: 'Basic ' + btoa( data: `${email}:${password}`),
      'cache-control': 'no-cache',
      'Content-Type':  'application/x-www-form-urlencoded',
    })
  };

  return this.http.get( url: this.url+"/login", options).pipe(
    tap( observerOrNext: (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.

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.
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.



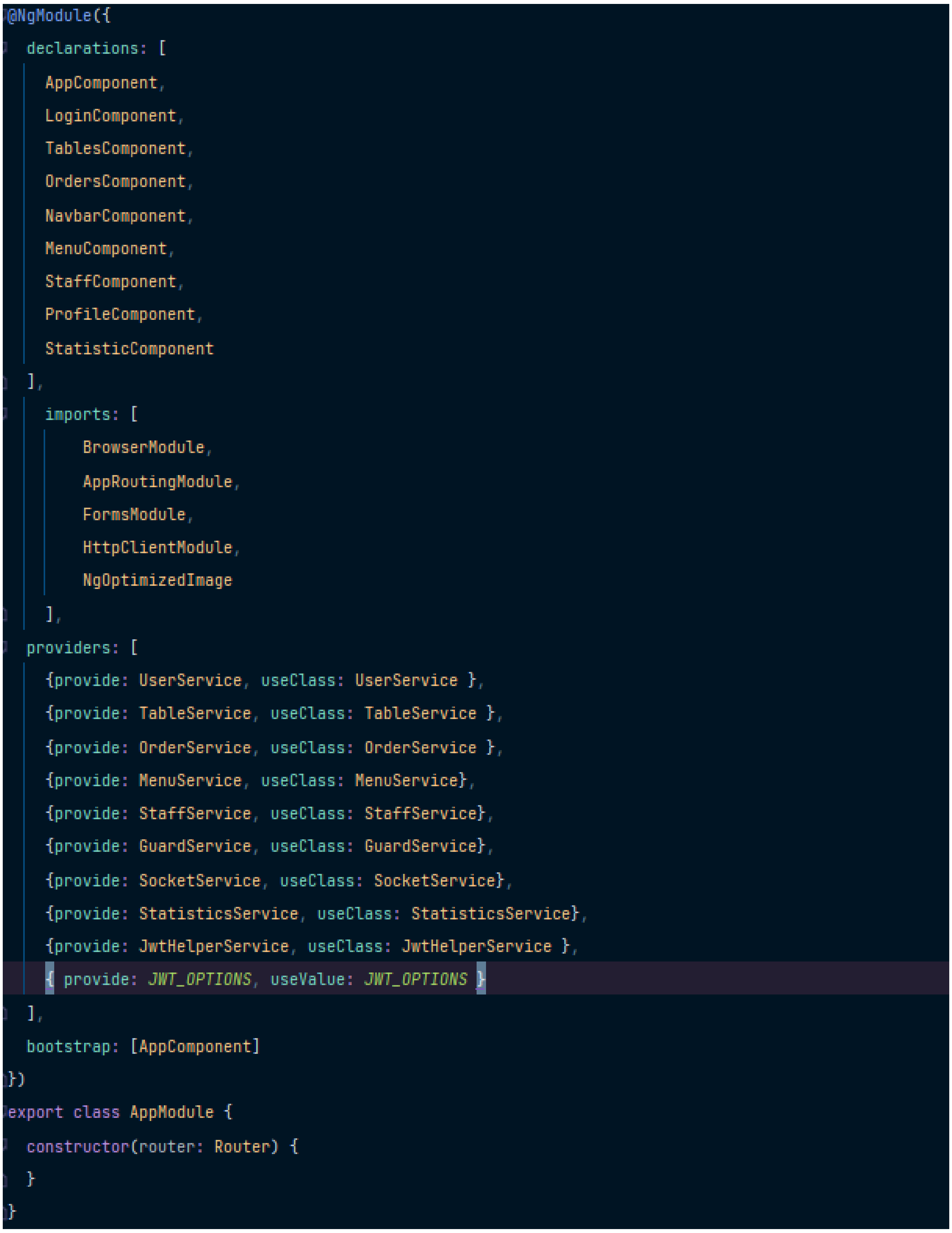
```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | boolean {
  if (state.url !== '/login') {
    if (!this.us.is_authenticated()) {
      this.router.navigate( commands: ["login"]);
      return false;
    }
    else if (!GuardService.routes[this.us.get_role()].includes(state.url.split( separator: '/')[1])) {
      this.router.navigate( commands: ["**"]);
      return false;
    }
    else return true;
  }
  else return true;
}
```

Signup

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, email, 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,
    OrdersComponent,
    NavbarComponent,
    MenuComponent,
    StaffComponent,
    ProfileComponent,
    StatisticComponent
  ],
    imports: [
        BrowserModule,
        AppRoutingModule,
        FormsModule,
        HttpClientModule,
        NgOptimizedImage
    ],
  providers: [
    {provide: UserService, useClass: UserService },
    {provide: TableService, useClass: TableService },
    {provide: OrderService, useClass: OrderService },
    {provide: MenuService, useClass: MenuService},
    {provide: StaffService, useClass: StaffService},
    {provide: GuardService, useClass: GuardService},
    {provide: SocketService, useClass: SocketService},
    {provide: StatisticsService, useClass: StatisticsService},
    {provide: JwtHelperService, useClass: JwtHelperService },
    { provide: JWT_OPTIONS, useValue: JWT_OPTIONS }
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
  constructor(router: Router) {
  }
}
```

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:

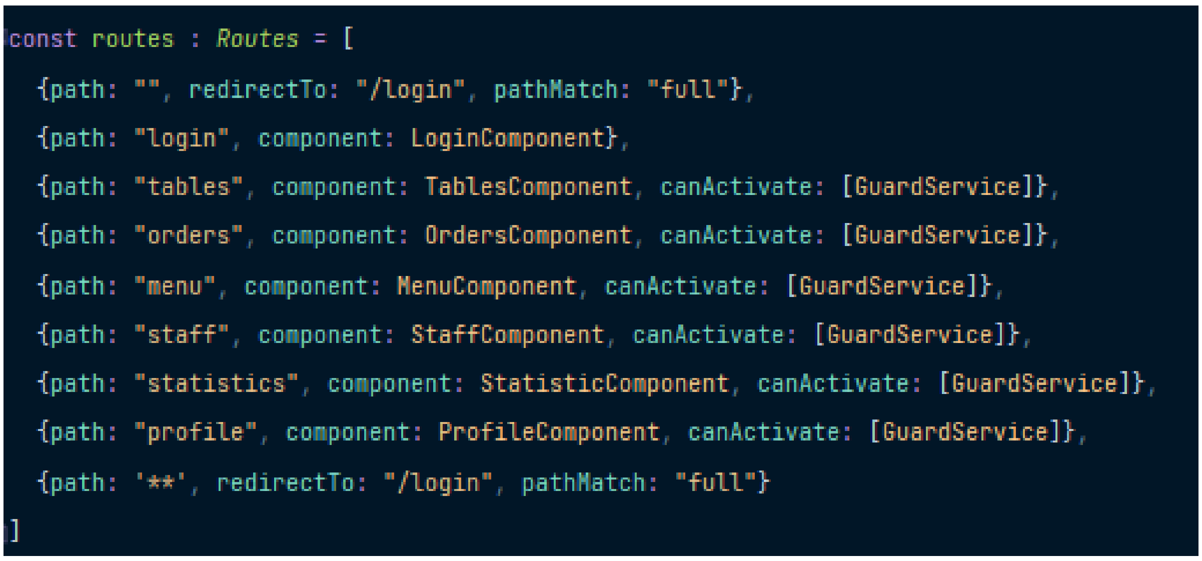
- **LoginComponent**: contains the form code and the login method.
- **NavbarComponent**: is used to implement the navbar that the users used to navigate inside the application and it changes based on the user role.
- **TablesComponent**: contains all the logic useful for the list of the tables visualization and the modification of them.
- **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 products with their information, adding them and removing them.
- **StaffComponent**: presents all the code used to handle the staff list and his modification, like adding or removing users.
- **StatisticsComponent**: contains all the logic to compute and display the users statistics.
- **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:

- ❖ **UserService**: implements the methods used to make APIs calls refers to the user logic.
- ❖ **TableService**: implements the methods used to make APIs calls refers to the table logic.
- ❖ **OrderService**: implements the methods used to make APIs calls refers to the order logic.
- ❖ **MenuService**: implements the methods used to make APIs calls refers to the products logic.
- ❖ **StaffService**: implements the methods used to make APIs calls refers to the staff logic.
- ❖ **StatisticsService**: implements the methods used to make APIs calls refers to the statistics logic.
- ❖ **SocketService**: implements the methods connect and get_update to implement the possibility to see changes in real time in some components.
- ❖ **GuardService**: implements the method canActive to check if a user is entering a route without being logging in or is entering a non-existent route.

Routes



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

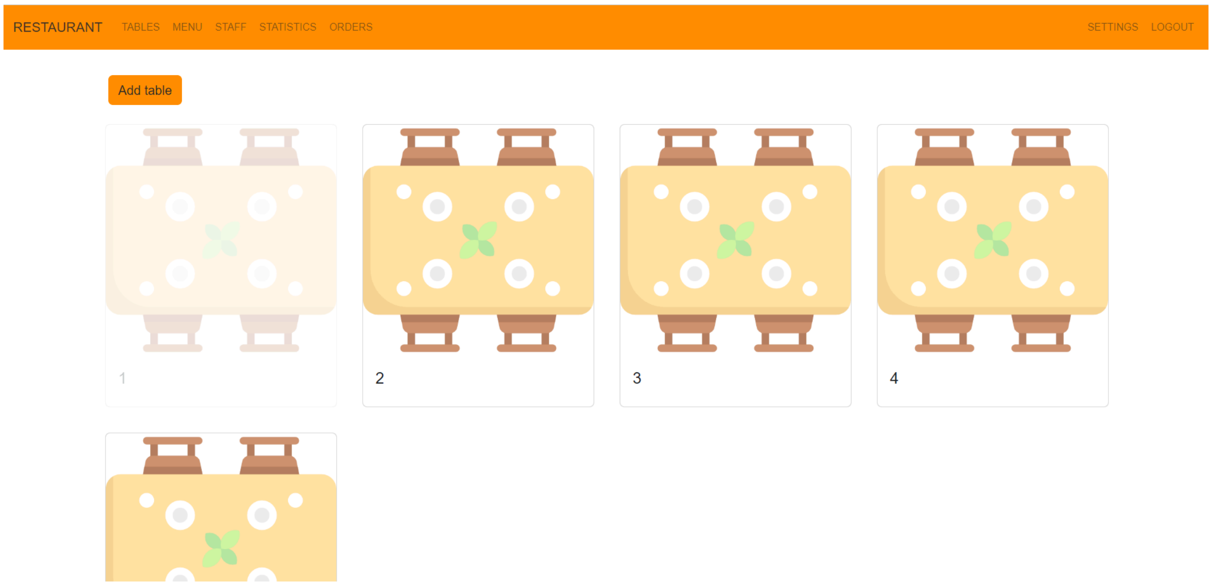
The login route is related to the access process, the staff, menu, orders and statistics routes display a general list of their data information, the profile route displays the user personal information, the ** is dedicated to all the non-existent routes and the remaining ones are dedicated to one specific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

Application Examples

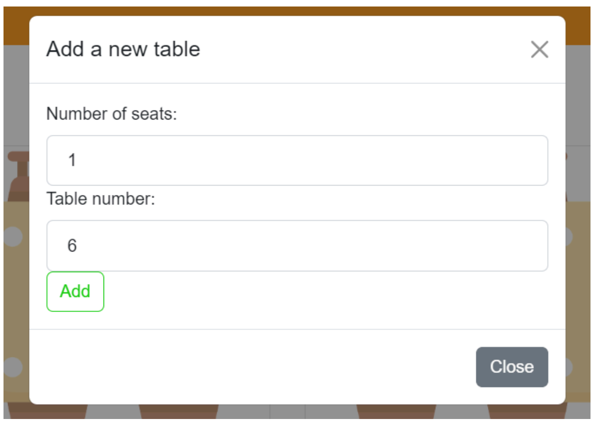
This section contains some examples of the application execution, divided by the user's role.

Cashier

After the login, the user is redirected to his dashboard. The cashiers are redirected to the tables page, where they can see occupied and free tables.



The cashier can add and update tables.


Add a new table

Number of seats:

1

Table number:

6

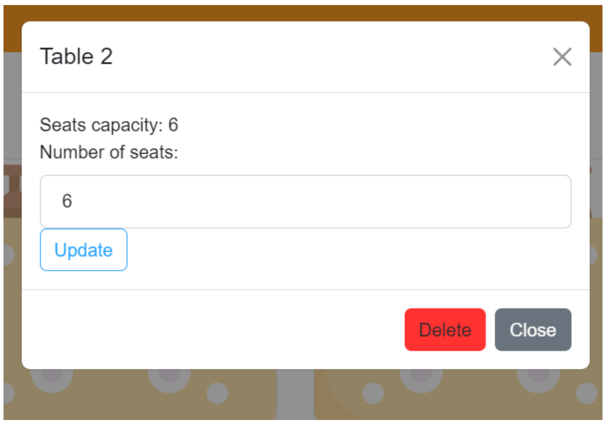


Table 2

Seats capacity: 6

Number of seats:

6

The cashier can see the orders of each table, print the receipt and do the bill.

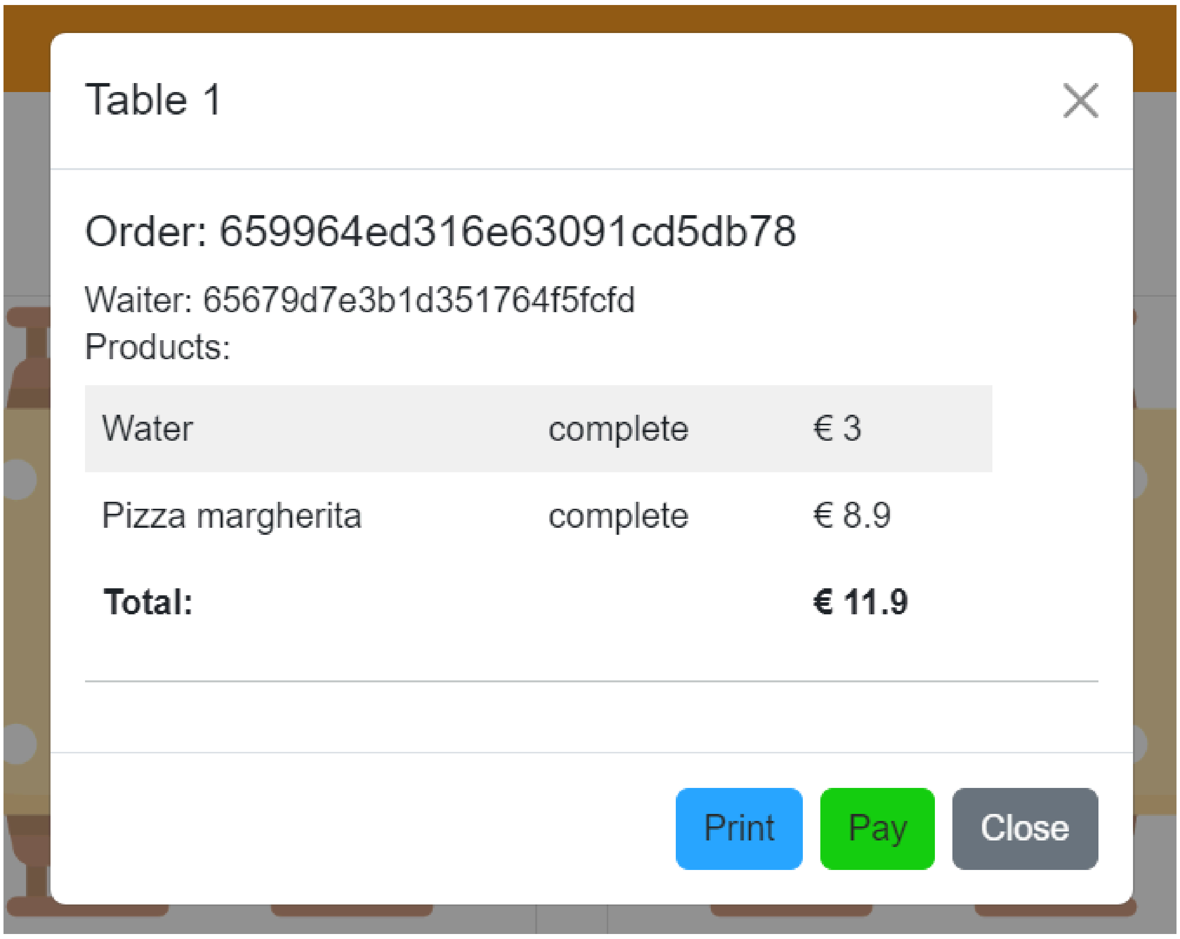


Table 1

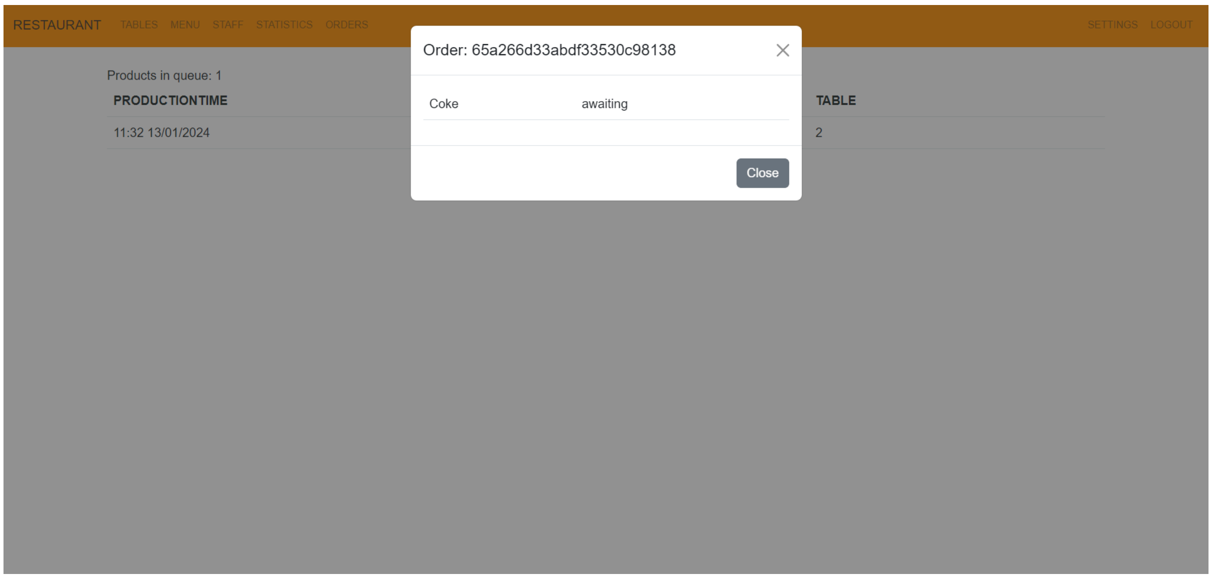
Order: 659964ed316e63091cd5db78

Waiter: 65679d7e3b1d351764f5fcfd

Products:

Water	complete	€ 3
Pizza margherita	complete	€ 8.9
Total:		**€ 11.9**

The cashier can see the queue of the orders and the status of the products.



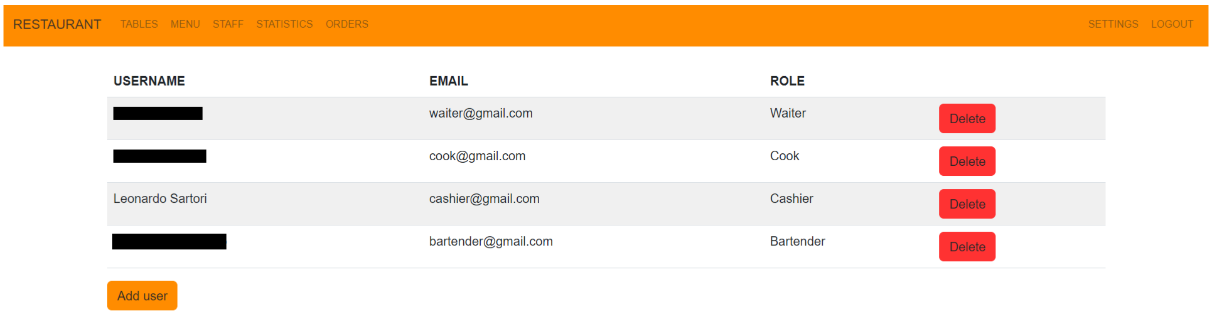
Products in queue: 1

PRODUCTIONTIME	TABLE
11:32 13/01/2024	2

Order: 65a266d33abdf33530c98138

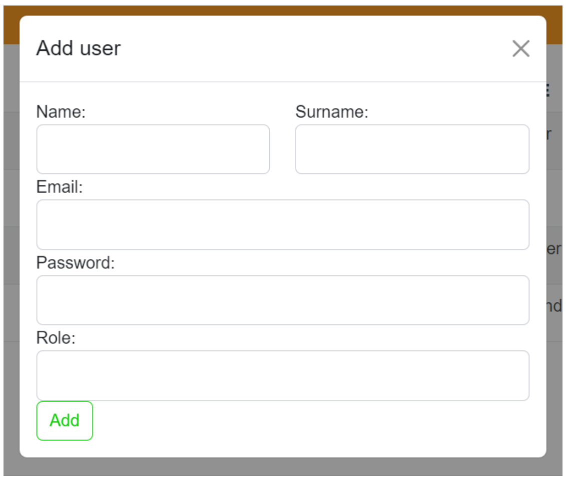
Coke	awaiting

The cashier is an admin and can manage the users: cashiers can add and remove them.

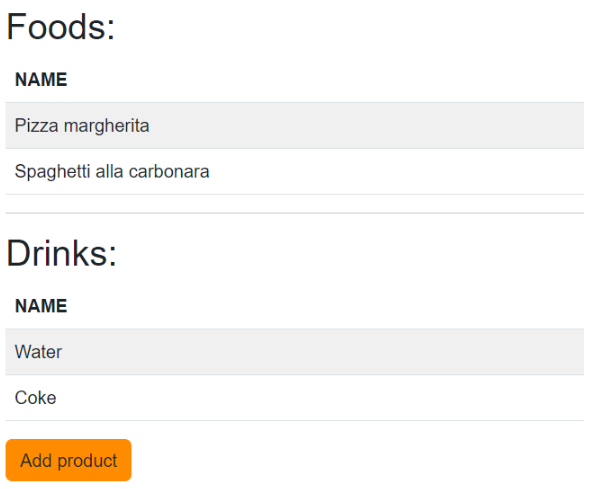


USERNAME	EMAIL	ROLE	
[redacted]	waiter@gmail.com	Waiter	Delete
[redacted]	cook@gmail.com	Cook	Delete
Leonardo Sartori	cashier@gmail.com	Cashier	Delete
[redacted]	bartender@gmail.com	Bartender	Delete

Cashiers, as admin, can add new users.



The cashier can also access the menu: cashiers can manage the products.

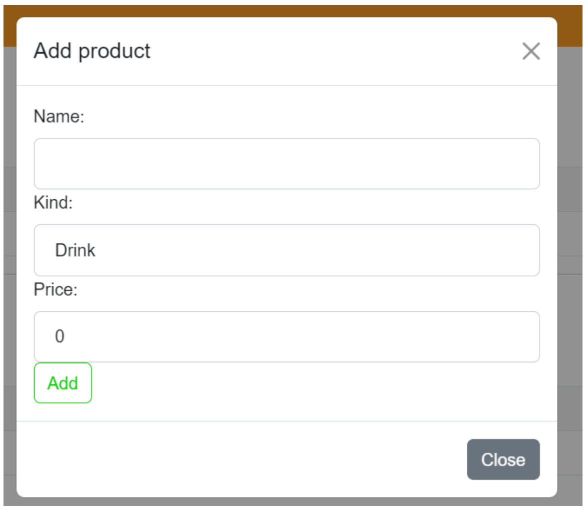


Foods:

NAME
Pizza margherita
Spaghetti alla carbonara

Drinks:

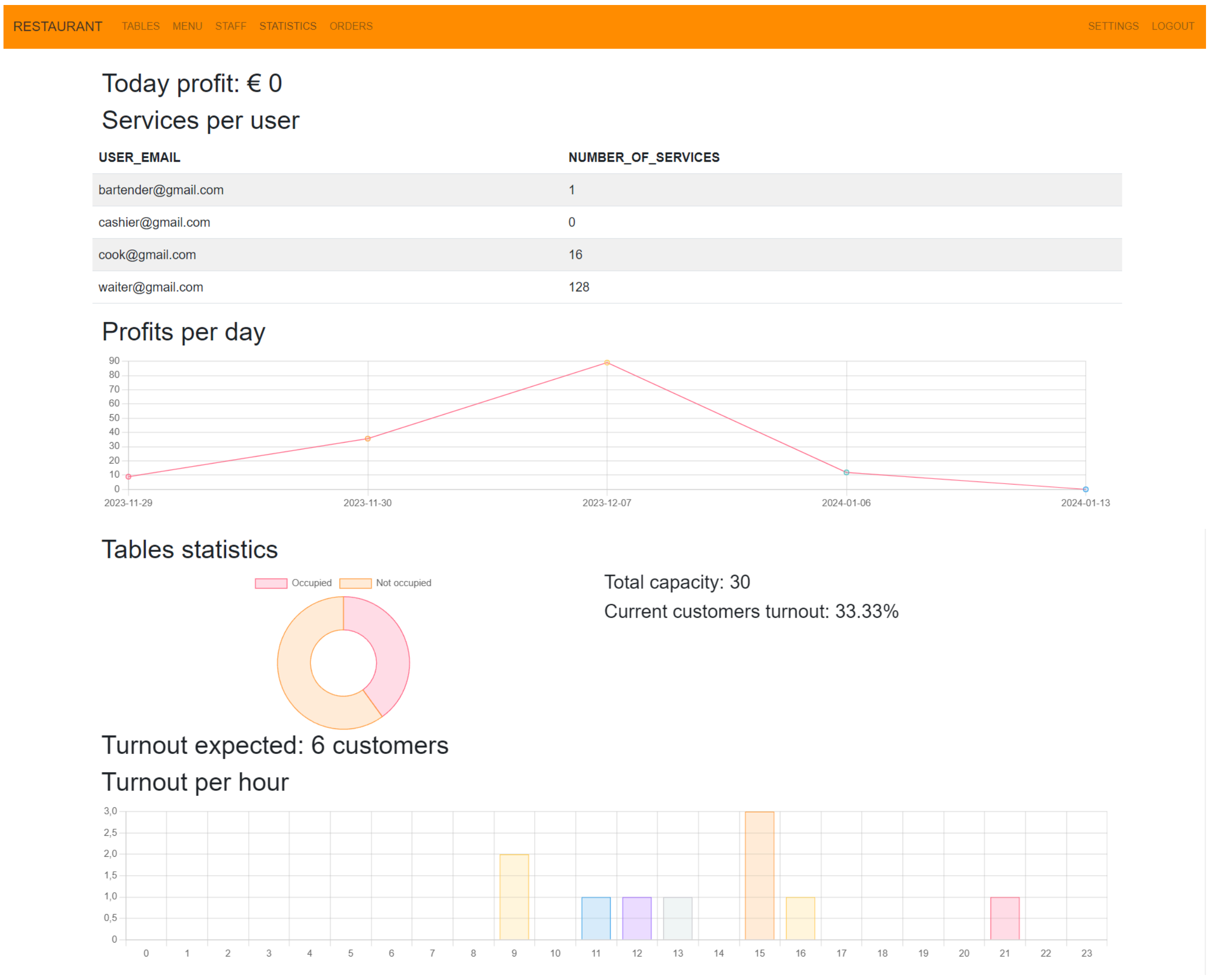
NAME
Water
Coke



Kind: Drink

Price: 0

The statistics page shows the profit of the day, services per user (this means how many products are served, by waiter, or prepared, by cook or bartender), the profits per day, table statistics (how many tables are occupied, customer turnout), prediction of customers and turnout per hour.



Today profit: € 0

Services per user

USER_EMAIL	NUMBER_OF_SERVICES
bartender@gmail.com	1
cashier@gmail.com	0
cook@gmail.com	16
waiter@gmail.com	128

Profits per day

Tables statistics

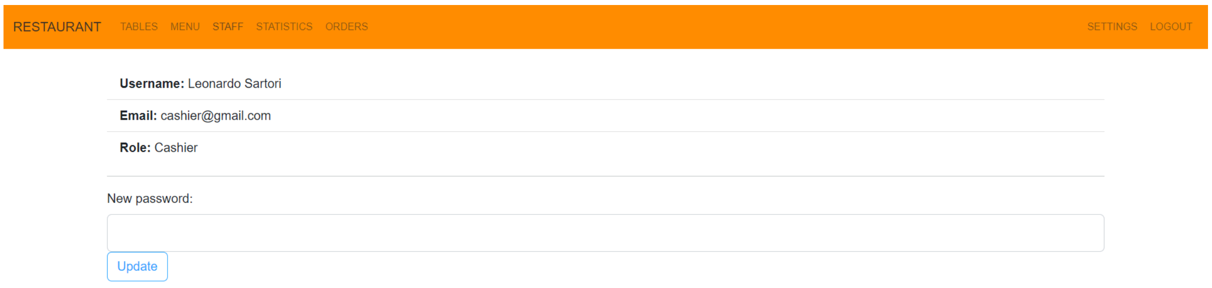
Total capacity: 30

Current customers turnout: 33.33%

Turnout expected: 6 customers

Turnout per hour

Finally, every user can update his password in the settings page.



Username: Leonardo Sartori

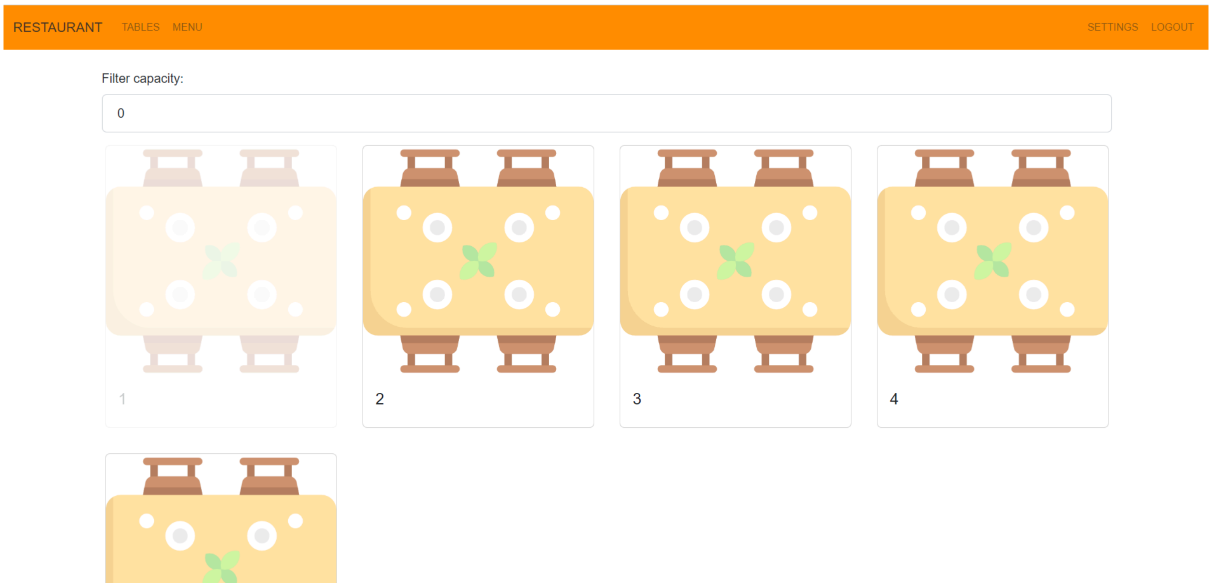
Email: cashier@gmail.com

Role: Cashier

New password:

Waiter

After the login, waiters are redirected to the tables page.



They can manage the orders of the tables: can occupy the table and create orders.
When a product is ready, the waiter can notify, that the product will be served. The bill can be done only if all the products are served.

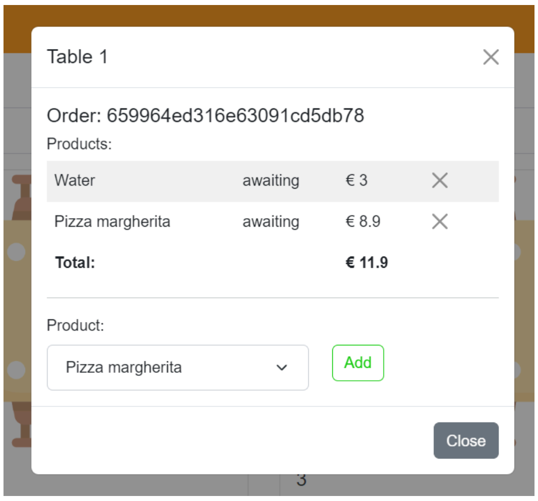


Table 1

Order: 659964ed316e63091cd5db78

Products:

Water	awaiting	€ 3	✕
Pizza margherita	awaiting	€ 8.9	✕
Total:		**€ 11.9**	

Product:

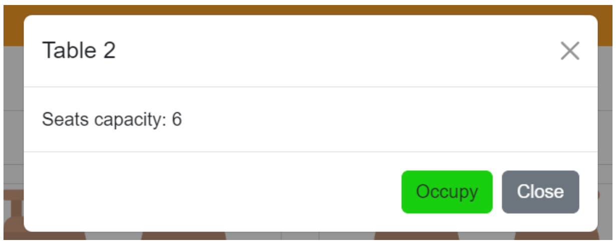


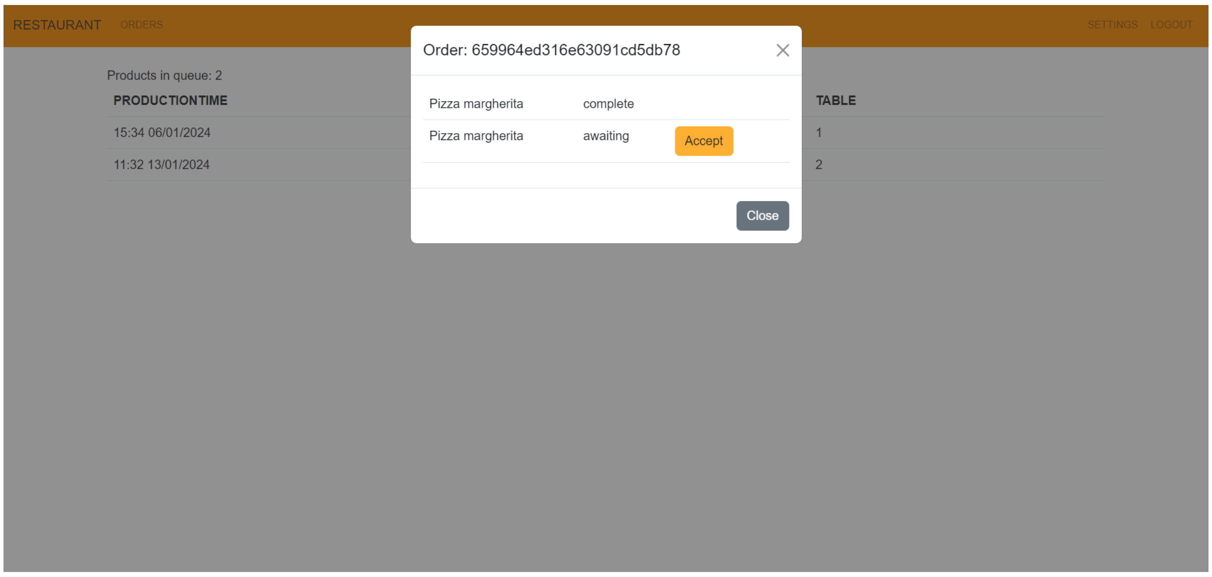
Table 2

Seats capacity: 6

Finally, waiters have access to the menu and, as the cashier, can manage it.

Cook and Bartender

Cooks and bartenders are similar roles. In fact, they have the same functionalities, but cooks work with food and bartenders work with drinks. This is the only difference.
After the login, they are redirected to the orders page where they can see the list of orders in the queue sorted by date and table.



Products in queue: 2

PRODUCTIONTIME	TABLE
15:34 06/01/2024	1
11:32 13/01/2024	2

Order: 659964ed316e63091cd5db78

Pizza margherita	complete	
Pizza margherita	awaiting	Accept

Cooks and bartenders can see only the orders that have at least one product that is not done. Specifically, cooks can see orders that have a *food* product to be done and bartenders can see orders that have a *drink* product to be done.