



**Project Of  
Tecnologie e Applicazioni Web  
2022/23  
University Ca' Foscari Venezia**

**Project Report CookHub  
1.0.0**



## Document Informations

Informazioni	
Deliverable	Progetto Tecnologie e Applicazioni Web
Data di Consegna	16/1/2024
Team members	<a href="#">STEFANO MASIERO,MAKSYM NOVYTSKYI</a>



## Index

<b>1. Introduction</b>	<b>4</b>
<b>2. Structure Of The Project</b>	<b>4</b>
<b>3. Description Of The Application</b>	<b>7</b>
<b>4. Conventions</b>	<b>8</b>
<b>5. Backend</b>	<b>8</b>
<b>5.1. MongoDB</b>	<b>8</b>
<b>5.2. Express.js</b>	<b>9</b>
<b>5.2.1. Structure Of The Backend</b>	<b>10</b>
<b>5.2.2. Models</b>	<b>11</b>
<b>5.2.3. Routes</b>	<b>11</b>
<b>5.2.3.1. Access Routes</b>	<b>11</b>
<b>5.2.3.2. List Of The Routes</b>	<b>12</b>
<b>5.2.3.3. Authentication</b>	<b>14</b>
<b>5.3. Redis</b>	<b>15</b>
<b>5.4. Swagger</b>	<b>16</b>
<b>5.5. Socket.IO</b>	<b>16</b>
<b>5.6. Backend Security in Production</b>	<b>17</b>
<b>6. Frontend</b>	<b>17</b>
<b>6.1. Structure Of The Frontend</b>	<b>17</b>
<b>6.2. Core</b>	<b>19</b>
<b>6.2.1. Components</b>	<b>19</b>
<b>6.2.2. Guards</b>	<b>19</b>
<b>6.2.3. Models</b>	<b>20</b>
<b>6.2.4. Services</b>	<b>20</b>
<b>6.3. Modules</b>	<b>20</b>
<b>6.3.1. Auth</b>	<b>20</b>
<b>6.3.2. Admin</b>	<b>21</b>
<b>6.3.3. Waiter</b>	<b>23</b>
<b>6.3.4. Production</b>	<b>25</b>
<b>6.3.5. Cashier</b>	<b>26</b>
<b>6.3.6. Analytics</b>	<b>27</b>
<b>6.4. Frontend in Production</b>	<b>29</b>
<b>7. Conclusions</b>	<b>31</b>



## 1. Introduction

This document reports the development process of a web app built using the MEAN stack, a technology stack consisting of MongoDB, Express.js, Angular, and Node.js. The scope of this application is the management of a restaurant system designed to handle table reservations and orders. The project utilizes a REST-style API as the backend, and an Angular frontend for user interaction.

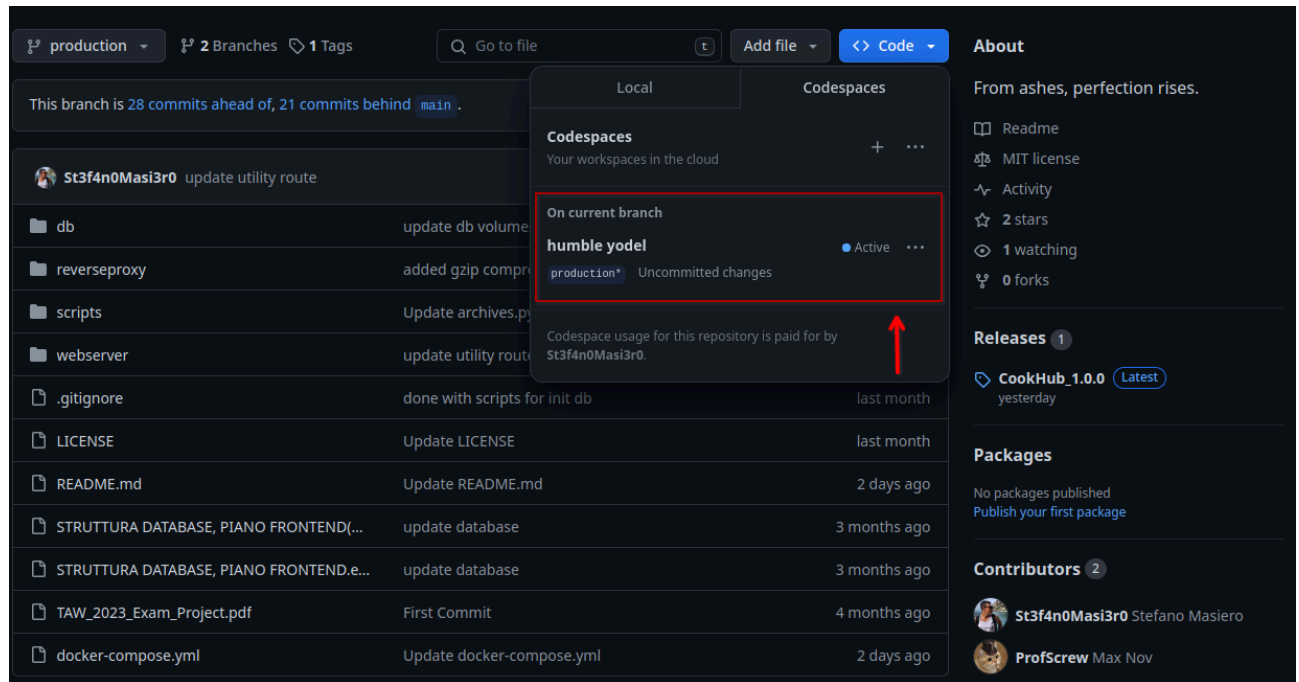
The project structure consists of the following components:

- ❖ **MongoDB:** A NoSQL database that stores the information about tables, orders, and customers.
- ❖ **Express.js:** A web framework based on Node.js that helps handle API requests and responses.
- ❖ **Angular:** A front-end framework based on Typescript, for building single-page applications (SPA).
- ❖ **Node.js:** A JavaScript runtime environment that executes Express.js and Angular code.
- ❖ **Redis:** A in-memory database, used as a caching mechanism for token blacklist and also for cache responses for the queries.
- ❖ **Nginx:** A high-performance, open-source web server and reverse proxy server used for serving our web applications.
- ❖ **Docker:** A containerization platform that helps us to package and manage the different services in an isolated environment.

## 2. Structure Of The Project

The team decided to develop all the project using Github, a famous versioning software that helped us track all the versions and bugs that we had fixed. One of the most important functionality that we had used of this software is the branch, we had divided the project into two main branches:

- ➔ **Development:** In this branch we had tried and experimented all the new features and after that we put the most important in the production branch. Is important to say that in this part we had put only the frontend for the web (no cordova or electron app).
- ➔ **Production:** In this branch we had put the final version of the application that includes also a mobile app for the android smartphone built with cordova and a desktop application built with electron. The production backend has been thought to run for example in a web hosting and for this reason has some little optimization. In our case we decided to host it in Github Codespaces that is free for students, but has disadvantages because after the start of the containers we have a limited amount of time ~15 minutes of execution and after that if there are no requests it automatically shuts down. For this reason if you want to test the mobile app or desktop app we have provided you in the Release folder either you notify us to start our hosting, or you should rebuild the apps (See section [6.4 Frontend in Production](#)).



All the branches use docker to serve the different services, we decided to create an Internal network to allow the different services communicating with each other and also an External network that is used for exposing the service for the end user to use.

An important choice that differentiates Production from Development is the decision to reduce the number of open ports exposed in the External Network to limit the attack surface.

#### Production ports open

Porta	Indirizzo inoltr...	Processo in ese...	Visibilità
80	https://humble-...	/usr/bin/docker-...	Public
443	https://humble-...	/usr/bin/docker-...	Public

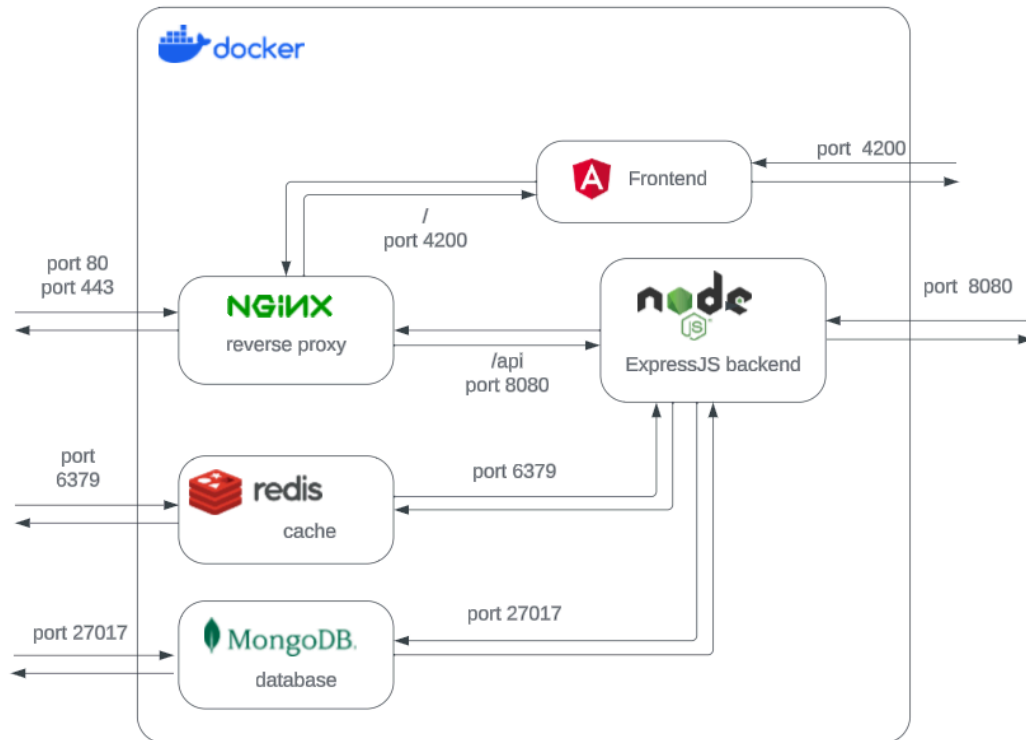
#### Development ports open

Porta	Indirizzo inoltr...	Processo in esecuzione	Visibilità
80	https://humble-...	/usr/bin/docker-proxy -proto tc...	Public
443	https://humble-...	/usr/bin/docker-proxy -proto tc...	Public
4200	https://humble-...	/usr/bin/docker-proxy -proto tc...	Public
6379	https://humble-...	/usr/bin/docker-proxy -proto tc...	Public
8080	https://humble-...	/usr/bin/docker-proxy -proto tc...	Public
27017	https://humble-...	/usr/bin/docker-proxy -proto tc...	Public

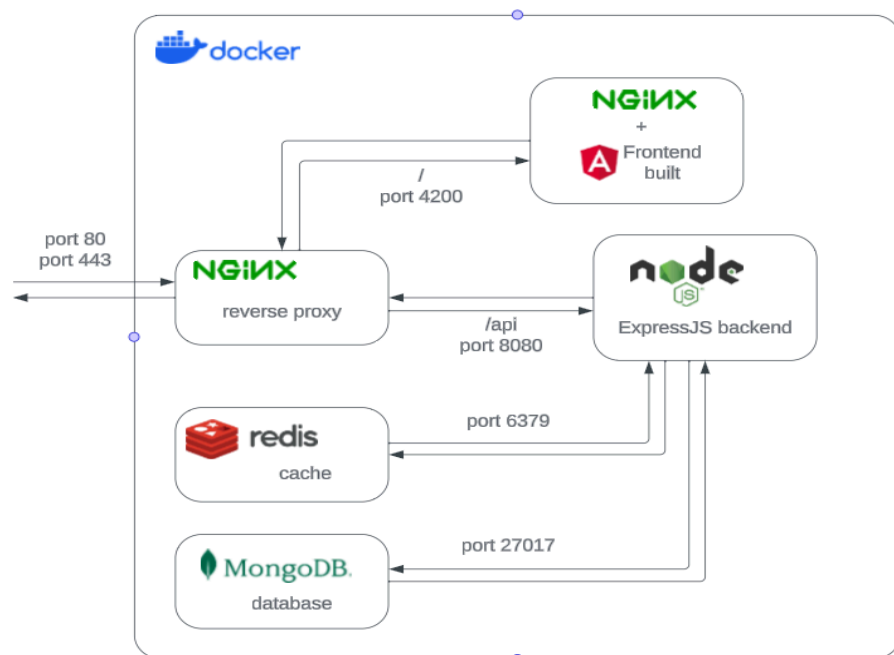


The images below show how the different components interact with each other in the two setup:

### Development Structure



### Production Structure





## How Interacts the Services

A brief explanation of how they work, i will explain the production, but the development branch services work in the same way (the only difference is the multistage docker where in the production the frontend is built and served from an nginx).

We have decided to implement a reverse proxy made with nginx that help us handle all the requests and forward to the correct services, for example all the requests that query for [127.0.0.1/api/](http://127.0.0.1/api/) are forwarded to the backend, and all each other are forwarded to the frontend. The application mainly works like this, the user will make a request to see the site at the url [127.0.0.1](http://127.0.0.1) , after the request has been processed the frontend will be returned. At this point, if the frontend has to carry out operations on the data (CRUD) it will query the backend based on the route it needs. The backend will process the request and evaluate whether it will be possible to use the cache (Redis) if the data had already been previously requested or whether it will have to request the data from the database (Mongodb).

Is important to say that all these interactions between services are carried out on the Internal docker network. Users can only access the services exposed via the reverse proxy thanks to the External Docker Network (ports 80 and 443 for the production branch).

## 3. Description Of The Application

The project aims to develop a program for efficiently managing a restaurant. This involves implementing a system that handles client orders, automatically dispatches them to the kitchen/bar, and notifies waiters when the orders are ready.

To augment the initial plan put forth by the professor, the team has opted to implement refinements. These modifications encompass a more granular breakdown of roles and the incorporation of a customizable system during the account creation process. Notably, users now have the flexibility to seamlessly combine multiple roles. For instance, an individual can concurrently hold positions as an Administrator, Cashier, and Analytics personnel, fostering a more versatile and adaptable system. Another crucial enhancement that we've chosen to incorporate involves implementing a comprehensive system to archive all previous orders. Additionally, we aim to meticulously maintain logs detailing the individuals responsible for creating, cooking, and serving each order, along with the respective timestamps.

Here are the roles implemented in the software and what they can do:

- ❖ **Admin:** manages all the main information about the restaurant, including the users, and can edit the following tables: Users, Categories, Recipes, Ingredients, Rooms, Tables.
- ❖ **Waiter:** can add orders, insert new dishes in the courses, receives notifications when a course is ready to be served, and confirms it.
- ❖ **Production:** this role includes both the kitchen and the bar, they see the working queue and mark the dishes the started and finished cooking/preparing, and when a course is completed they mark it.
- ❖ **Cashier:** sees a list of the orders, and after they are completed can cash them out, also sees a list of the tables and their status.
- ❖ **Analytics:** can view simple statistics about the restaurant, and view the history of the orders completed.



## 4. Conventions

Before starting describing the different services we have decided to adopt an important convention for naming the files and for distinguish them with more facility inside the backend and the frontend:

```
user      .      route      .      ts
name      .      type       .      extension
```

## 5. Backend

The backend includes the most important services necessary to make our application work with the data. Soon we will describe more in detail how each piece of this service works and how it is structured.

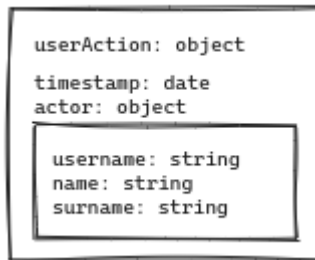
### 5.1. MongoDB

The beating heart of the application is the order data, and obviously to manage them we used a database namely [MongoDB](#), a versatile NoSQL database system, that stores data into JSON documents.

In the image below you can view all the structure of the documents and the name of the Collections that are present in the database



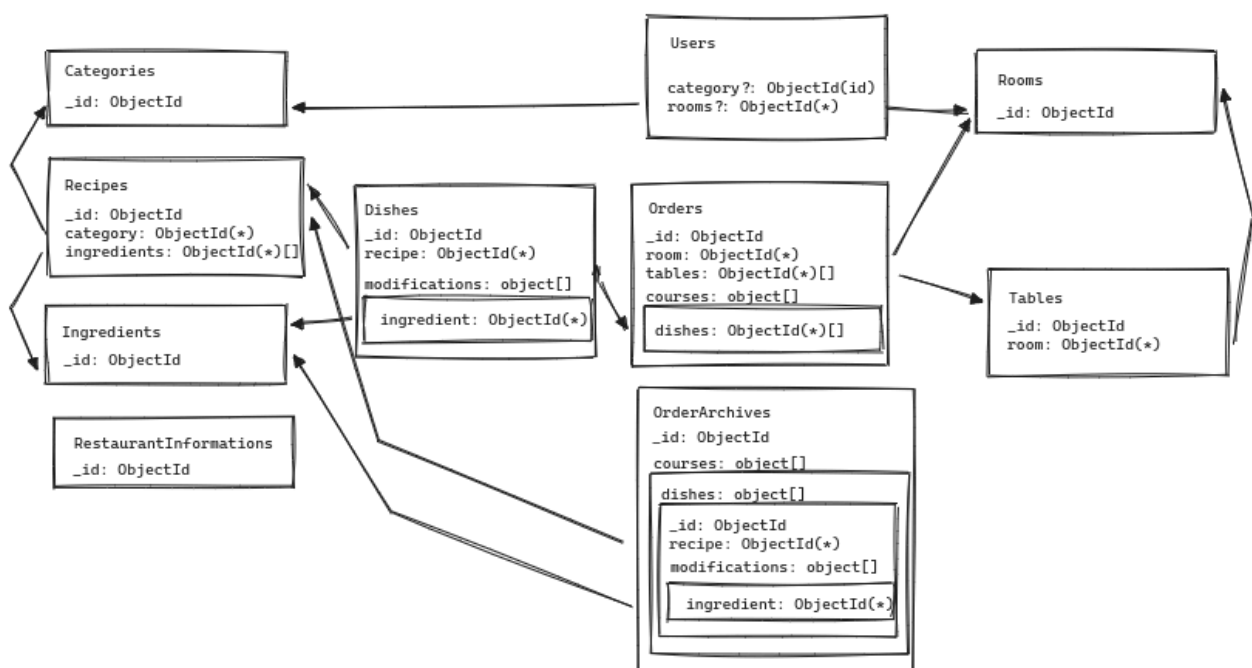




Note: When in the document the type is marked as userAction it is referred to this object that is used to make logs inside the database, saving the name of the user and a timestamp.

While designing the database structure we found ourselves discussing several times when to use linking or embedding in the documents, and this is ultimately the solution we proposed. As you can see very often we preferred linking rather than embedding solutions to avoid redundant information.

Here's a basic schema of the database collection that illustrates the relationships between collections:



**Security Note:** we know that it is more secure to create different users in the database and also manage permission at the database level for CRUD operation on every collection, but for this project we have decided to manage the permissions on the side of express.JS with the JWT token.

## 5.2. Express.js

[Express.js](#) is a Node.js web application framework that we have used to handle the different routes for our REST API. All this part is written in Typescript, and for understand how it works we need to look at the structure of the folders of this main part of the backend.

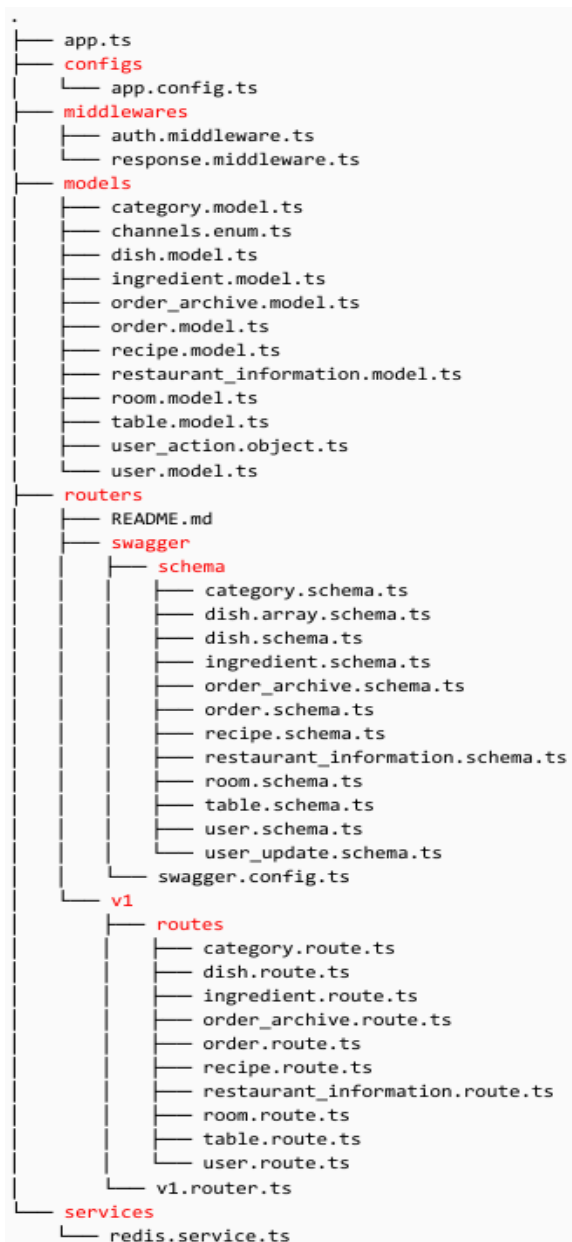


### 5.2.1. Structure Of The Backend

For working better we decide to structure the backend folder in this way:

- ★ **configs:** contains the configuration for the application like url, ports and some other configs
- ★ **services:** contains important services uses for interact with other components
- ★ **middlewares:** contains functions that allow to handle and manipulate http requests/responses
- ★ **models:** definitions of the model and schema used in the application for the database and express
- ★ **routers:**
  - **swagger:** API documentation, containing the schemas in OpenAPI format
  - **routes:** contains all the endpoints of the routes divided using SemVer

Below we show the most important files in the structures:





### 5.2.2. Models

In the models folder we defined all the models we needed to manage the data at the backend level, furthermore using the [mongoose](#) library we also automatically defined the structure of the documents that were saved in the database. Sometimes these models also have utility functions to control data or other functionality. Here is an example of the model for Room:

```
import {Schema, model} from 'mongoose';

export interface iRoom{
  _id: Schema.Types.ObjectId;
  name: string;
}

export const RoomSchema = new Schema<iRoom>({
  name: {type: String, required: true, unique: true},
},{
  versionKey: false,
  collection: 'Rooms'
});

export function verifyRoomData(room: iRoom): boolean {
  if (!room.name || room.name === '') return false;
  return true;
}

export const Room = model<iRoom>('Room', RoomSchema);
```

In this folder is also present a file "channels.enum.ts" that lists all the possible channels used in the Socket.IO (See section [5.5 Socket.IO](#))

### 5.2.3. Routes

In this folder we decide to implement the different routes, we use the standard [SemVer](#) for versioning our Rest API. Our version of API is version 1.0.0.

#### 5.2.3.1. Access Routes

For access to the routes and manage the data that are stored in the database the users must provide a valid JWT token. Below is showed an example of the payload contained in the JWT:

```
{
  "name": "admin",
  "surname": "admin",
  "username": "admin",
  "role": {
    "admin": true,
    "waiter": true,
    "production": true,
    "cashier": true,
    "analytics": true
  },
  "category": [],
  "room": [],
  "iat": 1705178740,
  "exp": 1705351540
}
```



Inside each route that requires specific permissions there is a piece of code like the one shown below that checks the permissions and if not allowed to enter refuses with the forbidden code!

```
const requester = req.user as iTokenData;

if (!requester || !requester.role || !requester.role.admin) {
  return next(cResponse.genericMessage(eHttpCode.FORBIDDEN));
}
```

#### 5.2.3.2. List Of The Routes

Before listing all the routes that we developed, we want to show the conventions that we decided to adopt for making our REST API.

The team tried to follow all the good practices said by the professor at lessons for developing a REST API, but also took a look at this nice [guide](#).

The method used are:

- **GET**: Used to retrieve documents, with an option (query) for filtering the data
- **POST**: Utilized for inserting new documents inside a collection.
- **PUT**: Used for updating a particular document in the collections.
- **DELETE**: Utilized for removing a specific document from a collection.

Another important part of our code is the response middleware that helps us return successful or error status codes as responses. Only on some routes we have decided to implement the pagination of the result, to speed up the query for example in the order\_archive where the documents are a lot.

Now we list all the endpoints with a brief description, the methods for access it and also which permissions are required:

#### Legend

Admin: **AD**   Production: **PD**   Cashier: **CS**   Analytics: **AN**   Waiter: **WT**   Everybody: **ALL**

Routes	Methods	Permissions	Descriptions
/categories	GET	ALL	get list of categories or single
/categories	POST	AD	create a new category
/categories/{id}	PUT	AD	update a category
/categories/{id}	DELETE	AD	delete a category
/dishes	GET	WT, PD, CS	get list of dishes or single



/dishes	POST	WT	create a new dish
/dishes/{id}/action/{type}	PUT	PD	modify status of a dish
/dishes/{id}	PUT	WT	update a dish
/dishes/{id}	DELETE	CS	delete a dish
/ingredients	GET	ALL	get list of ingredients or single
/ingredients	POST	AD	create a new ingredient
/ingredients/{id}	PUT	AD	modify an ingredient
/ingredients/{id}	DELETE	AD	delete an ingredient
/order_archives	GET	AN	get list of order_archives or single
/order_archives/{id}	POST	CS	create a new order_archives
/order_archives/{id}	PUT	X	not implemented
/order_archives/{id}	DELETE	AN	delete a order_archives
/user/login	POST	No token	Used for logging in, and getting the token
/user	GET	AD	get list of users or single user
/user	POST	AD	used by the admin to insert new users
/user/{username}	PUT	AD	used by the admin to modify a user
/user/{username}/	DELETE	AD	used by the admin for delete a user
/orders	GET	WT, PD, CS	get list of orders or single
/orders	POST	WT	create a new order
/orders/{id}/action/{choice}	PUT	WT, PD	modify status or details of order
/orders/{id}	DELETE	CS	delete a roder
/recipes	GET	ALL	get list of orders or single
/recipes	POST	AD	create a new recipe
/recipes/{id}	PUT	AD	update a recipe



/recipes/{id}	DELETE	AD	delete a recipe
/restaurant_informations	GET	ALL	get the restaurant information
/restaurant_informations	POST	AD	create restaurant information
/restaurant_informations/{id}	PUT	AD	update restaurant information
/rooms	GET	ALL	get list of rooms or single
/rooms	POST	AD	create a new room
/rooms/{id}	PUT	AD	update a room
/rooms/{id}	DELETE	AD	delete a room
/tables	GET	ALL	get list of tables or single
/tables	POST	AD	create a table
/tables/{id}/status/{type}	PUT	WT, CS	update the table status
/tables/{id}	PUT	AD	update a table
/tables/{id}	DELETE	AD	delete a table
/utility/resetCache	GET	AD	clear all the cache values (debug)

We highly suggest to use the swagger docs for understand better how the api works and which parameters require. (See section [5.4 Swagger](#))

#### 5.2.3.3. Authentication

In this section I want to show you the most important part of this application, the authentication middleware. This middleware manages the entire authentication process, from login to the creation phase of the JWT token which takes place via the **create\_token** function, up to the **authentication** carried out with the function of the same name and finally the management and verification of the token carried out with the **authorize** function. Furthermore, this part also contains some of the functions that allow managing the token blacklist (See section [5.3 Redis](#)).



The image below show the interface used for create the token:

```
export interface iTokenData {
  name: string,
  surname: string,
  username: string,
  role: iRole,
  category?: iCategory["_id"][],
  room?: iRoom["_id"][],

  iat?: number,
  exp?: number
}
```

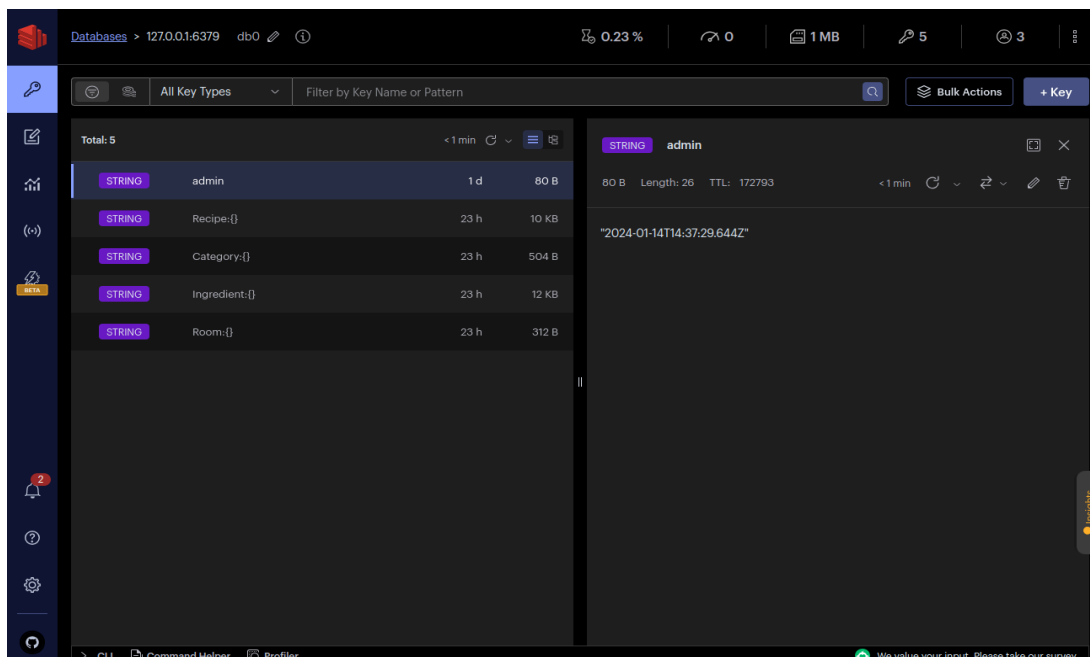
**Important:** All the settings inherent the token are inside the .env file

```
#TOKEN SECTION
JWT_SECRET = "SUPER_DUPER_SECRET_PASSWORD"
#Time expiration token (in hours) when createing
JWT_EXPIRATION = '48h' #Change to smaller value in production
#Time token ban when change info account (in seconds)
JWT_EXPIRATION_SECONDS = 172800
HASH_METHOD = 'sha512'
```

### 5.3. Redis

The team decided to optimize the performance of the backend using an in-memory database, [Redis](#), for caching the responses of the query on some certain collections, and for managing token blacklisting.

The image below shows an example of the values contained inside Redis for a token blacklist



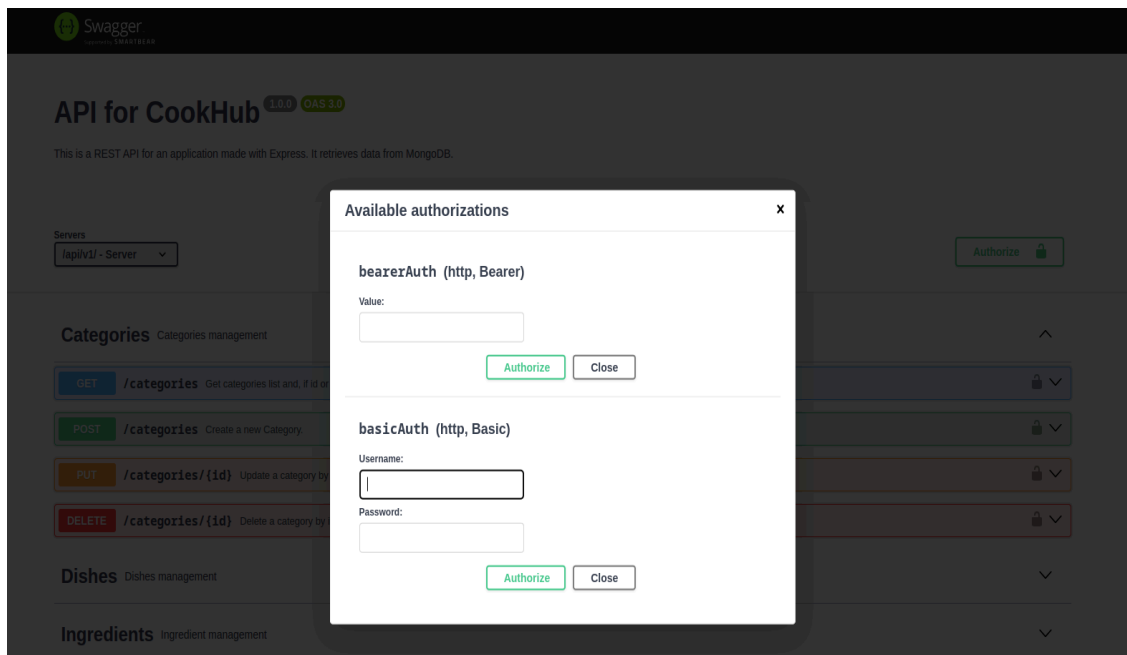


We created a singleton service that helps us use redis and manage the TTL (time to live) of each saved value. As I said before we decided to cache only collections where the documents don't change very often. If an update subsequently occurs in a collection whose documents were cached, then the old cached value is deleted and the new value is saved. The following collections are cached: Categories, Recipes, Ingredients, Restaurant Information, and Rooms.

The Redis cache also stores all the timestamps of the JWT tokens that have been blacklisted for us. Typically these are only marked as such when an administrator modifies a user, and so to maintain data integrity, tokens preceding the modifications timestamp are blacklisted in the cache until they expire.

**Note:** To prevent the application from working even without the caching system and therefore token blacklisting we decided to return to all the requests an error message that the Redis service is down if it is not working.

## 5.4. Swagger



The documentation created with [swagger](#) was developed by the team to quickly document and test the routes. To make it happen we used special tags in the Typescript code to document all the paths and we also had to recreate the template of our documents in openApi. To access the swagger documentation you can use the URL [127.0.0.1/api/v1/docs/](#). Next you need to click on the "Authorize" button, which opens a window (as shown in the image above) where you can enter your credentials using basic authentication. Next, use the `"/user/login"` path to get the token, which should be pasted in the bearerAuth section. Then you can test all the paths.

## 5.5. Socket.IO

We used [Socket.io](#) to ensure that clients always receive the latest information when data is updated. The client subscribes to a socket service, and if there are some changes in the backend, such as updates to data, a signal is sent to all clients via a special channel, so everyone stays updated. With the following snippet the backend sends the message that some changes have occurred to the data:





```
io.emit(eListenChannels.users, { message: 'User list updated!' });
```

After the clients receive the signal it requests updates using the normal API routes to download the updated data.

## 5.6. Backend Security in Production

The team decided to pay more attention to the security aspect in the production branch. In particular, the [helmet](#) library was added and all the ports of the docker containers that were not needed were also closed.

Another important aspect is that we have decided to provide the backend also through port 443, for use in a secure way using a self-signed certificate.

## 6. Frontend

The frontend, built with the [Angular](#) framework and using CSS, has been organized for easy development and a smooth user interface in a single-page application style. We also decided to add this library [Angular-Material](#) to enrich our application with a good style but at the same time also be [responsive](#), ready to adapt to mobile devices.

### 6.1. Structure Of The Frontend

The team before starting writing code tried to understand which should be the best structure for maintaining the project in the future and after viewing a lot of projects and guides decided to adopt the [LIFT](#) guideline that we found also in this good [guide](#).

The most important folder and modules are:

- **app**: It just takes responsibility loading all the other modules
- **core**: In this module we put all the functionality that are fundamental and viewable in all the frontend for example components, services, guards and models
- **modules**: Here are present some different folder, and every one implement a module with some components for a particular feature that the application can perform
- **assets**: In this folder are present all the static files that are useful for the frontend for example the images
- **environments**: Contains all the files with the settings for the Url of the backend

It is important to remember that we have decided to include a router in each module that manages the different routes, but globally there is also a router in the app module.

When a user use the frontend the module follow this workflow:

App module -> Core Module -> Module Feature



Below we show the most important files in the structures:

```
.
├── app
│   ├── app.component.css
│   ├── app.component.html
│   ├── app.component.ts
│   ├── app.module.ts
│   ├── app-routing.module.ts
│   └── core
│       ├── components
│       │   ├── error-page
│       │   ├── logo
│       │   ├── master-container
│       │   └── notifier
│       ├── core.module.ts
│       ├── core-routing.module.ts
│       ├── guards
│       │   └── auth.guard.ts
│       ├── models
│       │   ├── category.model.ts
│       │   ├── channels.enum.ts
│       │   ├── dish.model.ts
│       │   ├── ...
│       │   └── user.model.ts
│       └── services
│           ├── api.service.ts
│           ├── auth.service.ts
│           ├── database-references.service.ts
│           ├── page-data.service.ts
│           ├── page-info.service.ts
│           └── socket.service.ts
├── modules
│   ├── admin
│   │   ├── admin.module.ts
│   │   ├── admin-routing.module.ts
│   │   └── components
│   │       ├── categories
│   │       ├── dynamic-form
│   │       ├── ...
│   │       └── users
│   ├── analytics
│   ├── auth
│   ├── cashier
│   ├── production
│   └── waiter
├── assets
│   ├── images
│   │   ├── 403.jpg
│   │   ├── 404.jpg
│   │   └── logo.png
├── environments
│   ├── environment.development.ts
│   └── environment.ts
├── favicon.ico
├── index.html
├── main.ts
└── styles.css
```



## 6.2. Core

The core module contains the main features that allow the frontend to work, below we give a brief description of the various parts contained within it.

### 6.2.1. Components

The components includes in this modules are:

- ★ *logo*: this component shows a simple page with the logo of Cookhub, is our homepage.
- ★ *master-container*: this component contains the toolbar and the sidebar, we like to call it a container because the other components are rendered inside it while the application is running.
- ★ *notifier*: this component is called when the app wants to show backend success or error messages. To make this component we use the library [ngx-toastr](#)
- ★ *error-page*: this component shows the pages of error for example page not found or other errors.
- ★

### 6.2.2. Guards

We have opted to implement a Route Guard, which is an interface designed to intercept navigation on specific routes. This interface empowers us to manage user access by determining whether they should be granted entry to a particular page or not.

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';

import { AuthService } from '../services/auth.service';

export enum eRole {
  Waiter = 'waiter',
  Cashier = 'cashier',
  Production = 'production',
  Admin = 'admin',
  Analytics = 'analytics',
  Empty = '',
}

export const authGuard: CanActivateFn = (route, state) => {
  const auth = inject(AuthService)!;
  const router = inject(Router)!;

  const data = route.data['type'];

  switch(data) {
    case eRole.Empty: {
      return auth.isLogged() ? true : false;
    }
    case eRole.Waiter: {
      if (auth.isLogged() && auth.role['waiter'] == true) {
        return true;
      }
      break;
    }
  }
}
```



The code shown above allows you to verify which route the user wants to access, and based on this the information contained in the user's token are verified. If the user has the necessary permissions for access then he will be able to continue on the requested page, otherwise the user will see the forbidden error page.

### 6.2.3. Models

The models folder contains more or less the same files that are also present in the backend, because we use these models to manage data within our backend but also in the frontend. (See [5.2.2 Models](#))

### 6.2.4. Services

Inside the core modules are also contained all the services that the application needs, this is the list of the services:

- **auth.service.ts**: used to extract information from the token after authentication has occurred.
- **api.service.ts**: created by the team to be able to make requests and get responses from the backend more easily.
- **socket.service.ts**: service for listening and emitting signals using the socket.IO library.
- **page-info.service.ts**: we decided to use this service for setting up the name of the page in the toolbar, passing it from the modules components to the master-container.
- **page-data.service.ts**: service used to transfer generic data from one component to another, used for example by the waiters to transfer order data before inserting them into the database.
- **database-references.service.ts**: this service was done to increase the performance of the frontend and avoid making data requests where the documents have used linking to other documents. It is used on some light collections such as ingredients, categories, recipes... . These references are initialized only when the application starts (after login) and then only updated if such collections have changes via the socket service.

## 6.3. Modules

In this section we will briefly describe how the different modules work together and also show some images of the different components at work.

### 6.3.1. Auth

This module doesn't do anything special, it simply displays the login page and after the user enters the correct credentials, it redirects them to the homepage (logo components in the core module).

The image shows a login form with a purple border. At the top, it says 'Login' next to a user icon. Below that are two input fields: 'Username\*' and 'Password\*'. The password field has a small icon to toggle visibility. Below the password field is a checkbox labeled 'Remember me' which is checked. At the bottom right is a purple button labeled 'Login'.



### 6.3.2. Admin

We decided to create this module, for managing the pages of the admin (owner/s of the restaurant). The admin can modify in the database and in the frontend the data about: Users, Categories, Recipes, Ingredients, Information about restaurant, Rooms and Tables.

The team tried to develop components that are more reusable as possible, and for this reasons we built three different components and reuse this for built all the admin pages:

1. **dynamic\_form** to create a new document for a collection using different text fields or select with single or multiple choices.
2. **dynamic\_table** to display the existing documents of a collection in a table.
3. **dynamic\_table\_form** to edit a specific document that is present in the table with text fields and select with single or multiple choices.

If you look in the main or parent component, you'll discover models used to create dynamic components (the files with .standard extension). These models adhere to the "iDynamicForm," "iDynamicTable," and "iDynamicTableForm" interfaces, all of which are defined in the core models. These interfaces outline the structure of objects, including information like routes, columns, text fields, and more. They essentially specify what elements should be shown by the dynamic components. The dynamic components themselves are designed to take in an input object and then display the elements specified by the corresponding model. This setup allows for a modular and flexible approach, where the parent component defines the blueprint through these models, and the dynamic components execute and display the content based on the provided input.

The image below shows the object used for generating the dynamic form for ingredients:

```
modelInput: iDynamicForm = {
  route: '/ingredients',
  formName: 'newIngredient',
  textFields: [
    {
      name: 'name',
      label: 'Name',
      type: 'text',
      required: true,
      value: '',
    },
    {
      name: 'modification_price',
      label: 'Modification Price',
      type: 'number',
      required: true,
      value: '',
    },
    {
      name: 'modification_percentage',
      label: 'Modification Percentage',
      type: 'number',
      required: true,
      value: '',
    },
  ],
  arrayTextFields:
  {
    name: 'alergens',
    label: 'Allergens',
  }
};
```



Here you can see the output that is generated from the previous object

CookHub

Ingredients

0

Name	Modification Price	Modification Percentage
------	--------------------	-------------------------

Allergens

Allergens

Add

Submit

Here are some images of the Dynamic\_table and the Dynamic\_table\_form:

CookHub

Ingredients

0

Submit

Filter

name	modification_price	modification_percentage	alergens
Mozzarella cheese	3	15	dairy

Name	Modification Price	Modification Percentage
Mozzarella cheese	3	15

Allergens

dairy

Allergens

Add

Submit

Delete






Cherry tomatoes	2	10
Basil leaves	1	8
Balsamic glaze	4	12



### 6.3.3. Waiter

The waiter module groups together all the pages that allow you to manage the tables and the various orders attached to them. Furthermore, there is also a page that allows you to see the orders ready to be served.

Handling orders involves using the **order-form** and **orders-table components**. The waiter begins by entering information such as the number of guests, room, and table, and confirming it. Once confirmed, the order is added to the table. The waiter can then choose a specific order to retrieve customer information.

  CookHub  Orders  

### New Order

▼

▼

Rooms	Tables	Guests	Status
1	Table 11	1	waiting

1 – 1 of 1

< >

Items per page: 10 ▼



This action opens the **order-detail component**, displaying the details of the selected order (Screenshot on the left)). The waiter has the option to add courses and dishes to the order, leading to the **menu-selector component**. Here, all menu items are listed for selection (Screenshot on the right). Once the waiter finalizes the choices in the order-detail, confirming the order pushes all selected items to the database.

CookHub

Order Detail

0

Order detail of tables:

Table 11

Detail Order:

Room: 1

Table: Table 11

Guests: 1

Capacity: 10

Status: waiting

About to be added Courses:

Course: Number 1

Delete

Dish: *Classic Mojito* Rum, mint, lime, sugar, and soda water served over ice.

Price: 10€

Notes:

Course: Number 2

Delete

Dish: *Caprese Skewers* Fresh mozzarella, cherry tomatoes, and basil leaves drizzled with balsamic glaze.

Price: 8€

Notes:

Add Course

Submit Order

Adding a new Dish to the course:

<

Drinks

Appetizers

First C >

Classic Mojito

10€

^

Recipe Details

Item: Classic Mojito

Description: Rum, mint, lime, sugar, and soda water served over ice.

Price: 10€

Ingredients: Rum,Mint leaves,Lime,Sugar,Soda water

Add

Strawberry Basil Lemonade

8€

^

Espresso Martini

12€

^

Cucumber Mint Cooler

11€

^

Berry Blast Smoothie

9€

^

Dishes added to course:

Classic Mojito

10€

^

Send

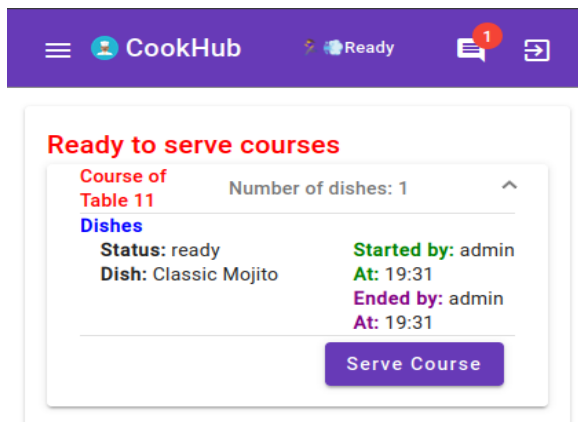
Taking and delivering orders in our system involves organizing them into courses. Once the kitchen or bar has prepared a course, a notification is sent to the waiters for service. An order is considered complete when all its courses have been served. When the kitchen or bar marks a course as completed in the system, a notification is automatically dispatched to all waiters via the notifier **core component**. Simultaneously, the notification counter on the top right corner increases by one (If you press on it is simply a shortcut for go to the ready component). To confirm the successful service of a course, waiters can navigate to the **ready component**, where they have the option to acknowledge and confirm the delivery of the specific course they handled.

TAW 2022/23 – Professor: Filippo Bergamasco



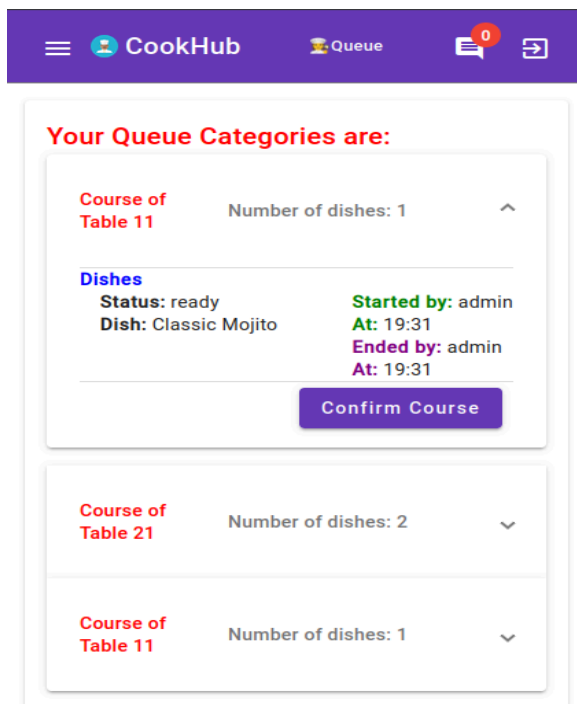


Here you can see the ready component at working



#### 6.3.4. Production

The production module, originally designated for the kitchen/bar, underwent a merger due to their similar functionalities. To differentiate between kitchen and bar tasks, the decision was made to implement categories assigned to user accounts. Each account is restricted to cooking recipes within their assigned category. For example, MonicaCook's account may be assigned the category "First Dishes". When she accesses her preparation queue, she will only see and be able to prepare foods within that specific category. The production module now consists solely of the **queue component**, operating on a first-in, first-out (FIFO) basis. This queue blends courses from various orders. Cooks and bartenders can select dishes to start and complete, and once all the dishes in a course are finished, they can confirm the course. Confirming a course triggers a notification to the waiter for serving that particular course in the dining area.





### 6.3.5. Cashier

The Cashier Module manages essential cashier-related tasks within a restaurant. Its primary functions include processing orders, archiving completed transactions, and providing real-time information on available tables. The process begins with the Cashier Module's **cashout component**, which maintains a list of active orders. Once an order is fulfilled, the cashier can review it by accessing the **order-detail component** (a concise summary of the order). Here, the cashier has the opportunity to confirm the details, prompting the transfer of the order from the active collection to the archive (orders that have been paid).

CookHub

Cashout

0

Order of Table 11

Number of courses: 1

Number of people: 1

Final Charge: 13€

General Info of Orders:

Guests: 1 people

Capacity: 10 people

Tables: Table 11

Room: 1

Service charge: 3€

Courses:

Course served by: admin At: 19:10

Course number: 1

Dish: Classic Mojito

Price of dish: 10€

Final Price: 13€

Cashout Order

CookHub

Cashout Detail Order

0

Cashout Order details

General Info of Restaurant:

Restaurant: CookHub

Address: Galileo

Email: cook@hub.com

Phone: 3456787654

IVA: 3436535634

General Info of Orders:

Guests: 1 people

Charge per Person: 3€

Service charge: 3€

Capacity: 10 people

Tables: Table 11

Room: 1

Courses:

Course number: 1

Dish: Classic Mojito

Price of dish: 10€

Final Price: 13€

Cashout Order



Additionally, the Cashier Module features a **tables component**, presenting a straightforward table status overview. This component efficiently displays the current status of each table, aiding the cashier in identifying and managing available tables.

Room Name	Table Name	Capacity	Status
1	Table 11	10	free
1	Table 12	10	free
1	Table 13	10	free
1	Table 14	10	free
1	Table 15	10	free

1 - 5 of 25

Items per page: 5

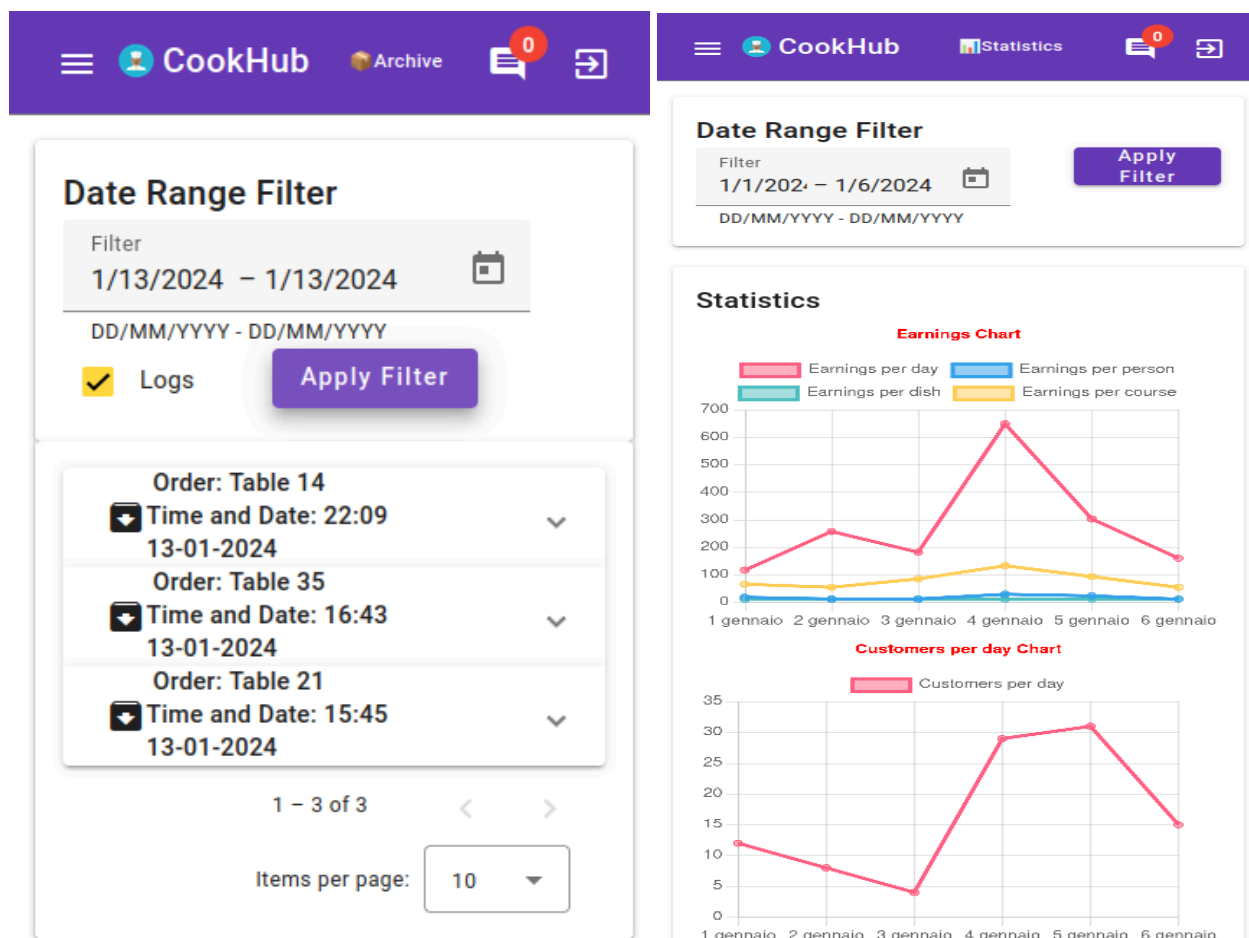
#### 6.3.6. Analytics

The Analytics role encompasses two primary components designed to facilitate order analysis and visualization:

1. **Archive Component:** This feature allows users to delve into the historical data of orders. Users can access and review past orders, filtering them based on specific dates or periods and also view the logs. This component provides a comprehensive archive of order history, enabling a detailed examination of the restaurant's past performance.
2. **Statistics Component:** The Statistics component utilizes [Chart.JS](#) to present straightforward visualizations. These visualizations offer insights into the overall performance of the restaurant, the efficiency of waitstaff, and production metrics. It's important to note that while these statistics are simple and may not be optimized for large data sets, they remain effective for gaining a quick understanding of key performance indicators. In situations with substantial data, there might be a slight delay in loading, typically taking a few seconds to process (despite the queries made to the backend being paginated).



Included are screenshots of these components in action, providing a visual representation of the analytical capabilities offered by this module.





## 6.4. Frontend in Production

As we already say in [Structure of the Project](#) we decided to create after ending to develop the Angular frontend a portable version of our application. For these reasons we wanted to build a version for smartphones and a version for Desktop, utilizing some framework like [Cordova](#) and [Electron.js](#) that simplify the process of creating .

The image below shows the directory where are contained the file for generate the portable app:

```
multi_platform_generate
├── cordova
│   ├── config.xml
│   ├── index.html
│   ├── logo.png
│   └── main.ts
├── electron
│   ├── index.html
│   ├── logo.ico
│   ├── main.js
│   └── package.json
├── general_config
│   ├── environment.development.ts
│   ├── environment.ts
│   └── socket.service.ts
├── generate_angular_multi_platform.sh
└── README.md
```

The only thing we had to do was to build the Angular frontend with **ng build** and then move the files obtained from the build into the framework and make some changes, for example changing the url for the backend API and socket service. To speed up this process and automate it, the team, after doing it manually the first time, wrote a simple bash script that would allow applications to be generated for the different platforms automatically.

If you want to try it you can follow the README.md inside the “multi\_platform\_generate” folder to build the application with your specific hosting. Obviously after creating the app before starting it you need to start the backend in your hosting!

If you don't want to compile the app you can use the version generated by us that you find in this link [Release](#) (**Important:** if you follow this route you must ask us to start the backend or you will not be able to use the application).



Below you can view a screenshot of the application for Linux:

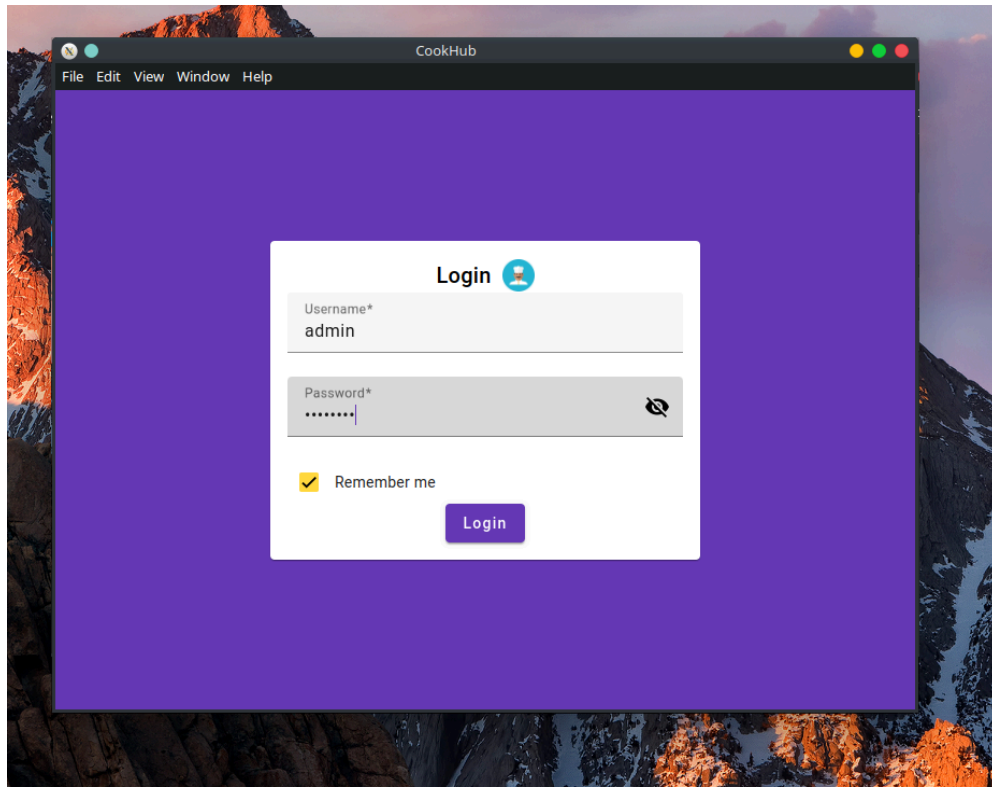


Image show the android version of the application:





## 7. Conclusions

We built CookHub using the MEAN technology stack, which was new to all of us. After some learning, we didn't encounter any major problems in developing the app and we learned a lot of things during the development process.

CookHub is now a simple prototype, missing some features needed for a real-world app. We plan to improve the performance and internal logic of some components and also the backend in upcoming releases. Furthermore, working with Docker to develop the different services was fantastic, because it allowed us to make the different parts interact with each other in a simple and orderly way. In short, building CookHub helped us use what we learned in class in a practical way and at the same time made us develop new skills in the field of web development and where it is important to work in a team to complete a fairly large project.