



**Progetto di
Tecnologie e Applicazioni Web
2022/23
Università Ca' Foscari Venezia**

**Project Report CookHub
1.0
by
Maksym Novytskyy**



Document Informations

Informazioni	
Deliverable	Report Progetto TAW
Data di Consegna	16/01/2024
Team members	STEFANO MASIERO , MAKSYM NOVYTSKYY



Indice

1. Introduction	4
2. Installation and Startup	5
3. High-level application description	6
4. Database design	7
5. Backend	9
5.1. Structure of the application	9
5.2. Models	11
5.3. Routes	12
5.4. Access to routes	13
5.5. Authentication	15
5.6. Redis	16
5.7. Swagger	16
5.8. Socket.IO	17
6. Frontend	17
6.1. Structure of the Application	17
6.2. Core Module	19
6.2.1. Models	19
6.2.2. Guards	19
6.2.3. Services	20
6.3. Feature Modules	20
6.3.1. Auth	20
6.3.2. Admin	21
6.3.3. Waiter	23
6.3.4. Production	25
6.3.5. Cashier	26
6.3.6. Analytics	28
7. Conclusions	29



1. Introduction

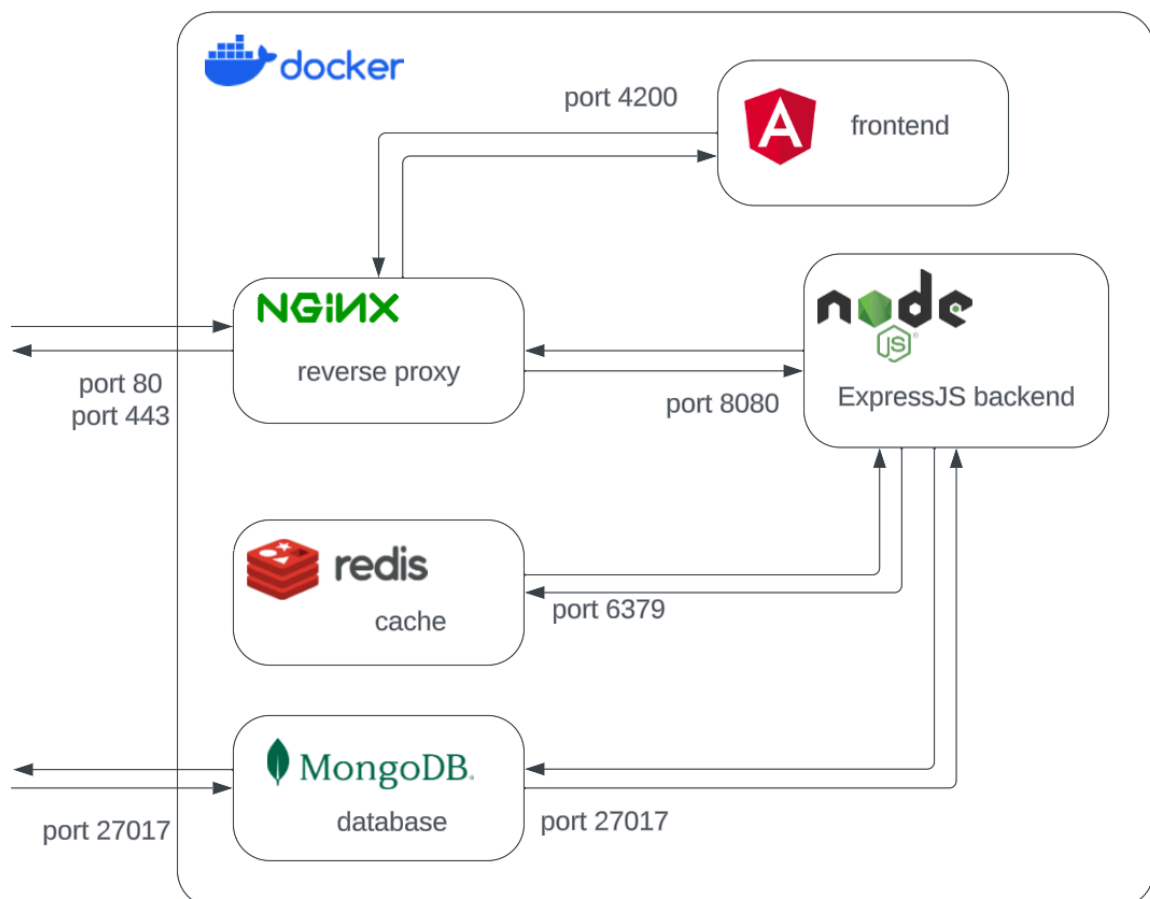
The scope of this document is to document and give light to the development process of the Technology and Web Application project.

The project consists in the development of a web application consisting of a REST-style API backend and a SPA Angular frontend. Also it should utilize a specific set of technologies referred as MEAN:

- MongoDB (a NoSQL database)
- Express.js (a web application framework for Node.js)
- AngularJS or Angular (a frontend JavaScript framework)
- Node.js (a JavaScript runtime environment)

The theme of the project is to create an application used in a restaurant environment to manage tables and orders.

Here is the structure of the project showing the components involved and how they interact with each other.





2. Installation and Startup

The installation process is simple, you will only require docker installed and python to fill up the database with data.

After you installed the prerequisites you will have to just to use docker-compose command to start up the application by typing the following command:

```
docker-compose up
```

After the app has started up, it is essential to have data in the database for the review of the app. You can do that by moving to the folder scripts:

```
cd scripts
```

Next step is to make you have all the right python libraries installed to do that use the requirements.txt file:

```
pip3 install -r requirements.txt
```

Last step is to start the script.

```
python3 init.py
```

The script fills the database with testing data like accounts of users, categories, recipes, ingredients and archive of orders.

Note: the accounts created in the application can be found in the Readme.md file. But if you want to use an account to test all the features of the application you can use the account with the following information:

```
username: admin password: password
```

After that you can just browse to the localhost or the address of the machine that the app is running on.



3. High-level application description

The goal of the project is to design a program to help manage a restaurant by implementing a system to handle orders from clients and automatically send them to the kitchen/bar and notifying the waiters that the orders are ready.

The team decided to make some changes to the original concept presented by the professor by breaking down the roles and also offering a customizable system when creating an account by combining multiple roles together.

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

- **Admin:** manages all the main information about the restaurant, and can edit the following tables: Users, Categories, Recipes, Ingredients, Rooms, Tables.
- **Waiter:** can add orders, insert recipes in the menu, 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.
- **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 orders.

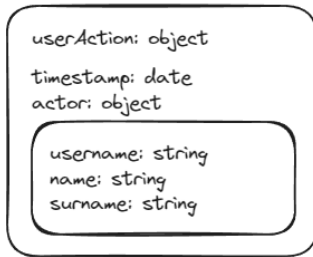


4. Database design

In order for the application to function properly we need to store the information generated by the restaurant in a database. For this purpose we will be using MongoDB, a versatile NoSQL database system, that stores data in a format similar to JSON documents.

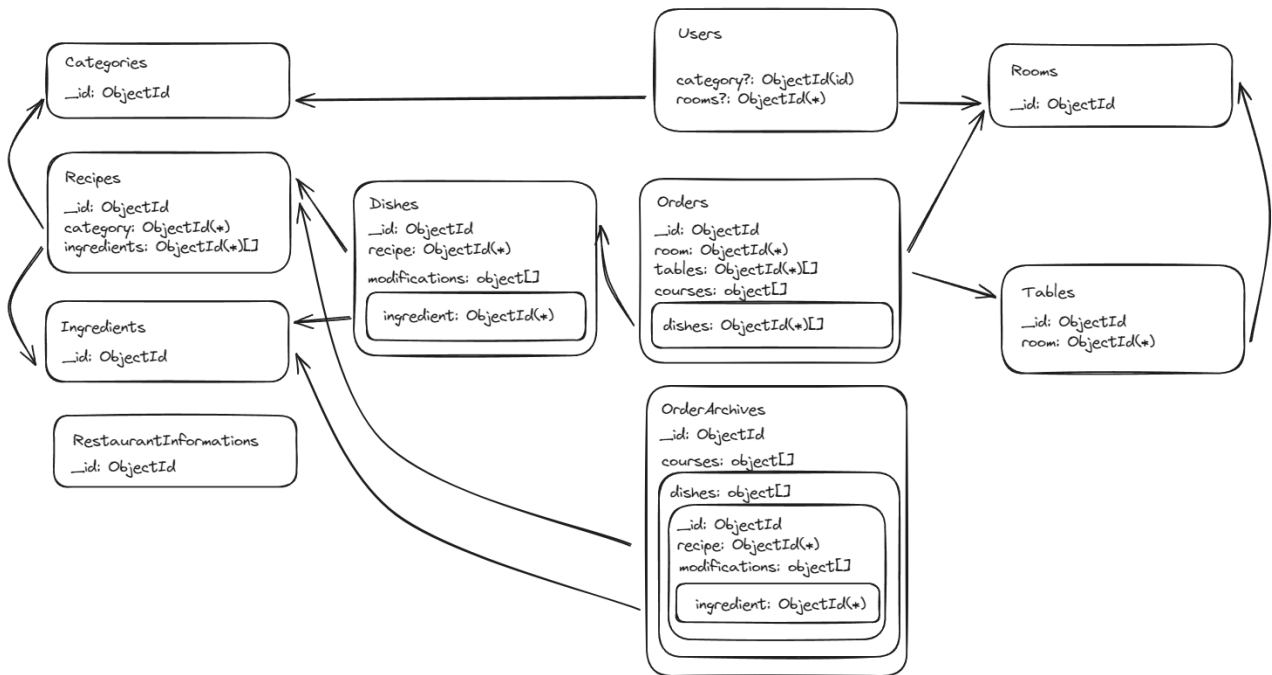
Here are all the collection used in the project with they internal structure





Note: When in the collection 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.

However, for the database to operate efficiently, establishing connections between collections is essential to minimize redundant data. Here's a basic schema of the database collection that illustrates these relationships:



Unfortunately, the limitations inherent in the community version of MongoDB restrict certain features. As a result, the management of data types, triggers, and collection permissions will be handled exclusively by the backend.



5. Backend

The backend, powered by Node.js and using the Express.js framework, supports a solid architecture ensuring efficient development and smooth handling of HTTP requests.

5.1. Structure of the application

In order to have a more comprehensive backend and to have easy and fast access to the desired part it is good practice to organize the files into a good folder structure. Here is the structure implemented in the backend:

- *configs*: contains the configuration for the application like links, ports and keys
- *services*: encapsulates the functionality for a specific task
- *middlewares*: intermediates between client and server
- *models*: definitions of structures used in the application of the database models
- *routers*:
 - *swagger*: API documentation, containing the schemas
 - *routes*: contains all the endpoints of the routes

Additionally the files follow the following structure in order to distinguish them and not create confusion:

user	.	route	.	ts
name	.	type	.	extension

Below will be listed the full tree of the backend files:



```
.
├── app.ts
├── configs
│   └── app.config.ts
├── middlewares
│   ├── auth.middleware.ts
│   └── response.middleware.ts
├── models
│   ├── category.model.ts
│   ├── channels.enum.ts
│   ├── dish.model.ts
│   ├── ingredient.model.ts
│   ├── order_archive.model.ts
│   ├── order.model.ts
│   ├── recipe.model.ts
│   ├── restaurant_information.model.ts
│   ├── room.model.ts
│   ├── table.model.ts
│   ├── user_action.object.ts
│   └── user.model.ts
├── routers
│   ├── README.md
│   ├── swagger
│   │   ├── schema
│   │   │   ├── category.schema.ts
│   │   │   ├── dish.array.schema.ts
│   │   │   ├── dish.schema.ts
│   │   │   ├── ingredient.schema.ts
│   │   │   ├── order_archive.schema.ts
│   │   │   ├── order.schema.ts
│   │   │   ├── recipe.schema.ts
│   │   │   ├── restaurant_information.schema.ts
│   │   │   ├── room.schema.ts
│   │   │   ├── table.schema.ts
│   │   │   ├── user.schema.ts
│   │   │   └── user_update.schema.ts
│   │   └── swagger.config.ts
│   └── v1
│       ├── routes
│       │   ├── category.route.ts
│       │   ├── dish.route.ts
│       │   ├── ingredient.route.ts
│       │   ├── order_archive.route.ts
│       │   ├── order.route.ts
│       │   ├── recipe.route.ts
│       │   ├── restaurant_information.route.ts
│       │   ├── room.route.ts
│       │   ├── table.route.ts
│       │   └── user.route.ts
│       └── v1.router.ts
└── services
    └── redis.service.ts
```



5.2. Models

In the models folder are defined the interfaces of the collections of the database, then the corresponding schemas, model, verification of data function, and if required some additional functionality related to the collection. Here is an example of the models are defined:

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

export interface iCategory {
  _id: Schema.Types.ObjectId;
  name: string;
  color: string;
  order?: number;
}

export const CategorySchema = new Schema<iCategory>({
  name: {type: String, required: true},
  color: {type: String, required: true},
  order: {type: Number, required: false, unique: true},
},{
  versionKey: false,
  collection: 'Categories'
});

export function verifyCategoryData(cat: iCategory): boolean {
  if(!cat.name || cat.name === '') return false;
  if(!cat.color || cat.color === '') return false;
  return true;
}

export const Category = model<iCategory>('Category', CategorySchema);
```



5.3. Routes

To facilitate data exchange between the server and client, a REST-style API has been implemented. Routes are structured based on the database collections, adhering to the following standard:

- **GET**: Used to retrieve information, with an option to input (query) an ID for specific data retrieval.
- **POST**: Utilized for inserting information into the relevant collection.
- **PUT**: Used to modify a particular record within a collection.
- **DELETE**: Utilized for removing a specific record from a collection.

However, there are exceptions where more specific routes have been implemented to perform certain actions. This approach aims to boost security by restricting certain users from having full privileges over particular information. Below, the implemented backend routes will be displayed. For further information about these routes, a Swagger documentation has been implemented. Can be accessed by following this link "<http://localhost/api/v1/docs/>"

User: /users/

- Endpoint: /login
method: POST
description: used for logging in, and getting the token
- Endpoint: /
method: POST
description: used by admin to insert new user
- Endpoint: /
method: GET
description: get list of users or single
- Endpoint: /{username}
method: PUT
description: modify user
- Endpoint: /{username}
method: DELETE
description: delete a user

RestaurantInformations: /restaurant_informations/

- Endpoint: /
method: GET
description: retrieve restaurant information
- Endpoint: /
method: POST
description: create restaurant information
- Endpoint: /{id}
method: PUT
description: modify restaurant information

Ingredients: /ingredients/

- Endpoint: /
method: GET
description: retrieve list of ingredients
- Endpoint: /
method: POST
description: create a new ingredient
- Endpoint: /{id}
method: PUT
description: modify ingredient by id (shadow)
- Endpoint: /{id}
method: DELETE
description: delete ingredient by id (shadow)

Rooms: /rooms/

- Endpoint: /
method: GET
description: retrieve list of rooms
- Endpoint: /
method: POST
description: create a new room
- Endpoint: /{id}
method: PUT
description: modify room by id
- Endpoint: /{id}
method: DELETE
description: delete room by id

Tables: /tables/

- Endpoint: /
method: GET
description: retrieve a list of tables
- Endpoint: /
method: POST
description: used by admin to create new table
- Endpoint: /{id}/status/{type}
method: PUT
description: update status of table by id
- Endpoint: /{id}
method: PUT
description: modify table by id
- Endpoint: /{id}
method: DELETE
description: delete table by id

Orders: /orders/

- Endpoint: /
method: GET
description: retrieve list of orders
- Endpoint: /
method: POST
description: create a new order
- Endpoint: /{id}/action/{choice}
method: PUT
description: modify status or details of order
- Endpoint: /{id}
method: DELETE
description: delete order by id

Categories: /categories/

- Endpoint: /
method: GET
description: retrieve list of categories
- Endpoint: /
method: POST
description: create a new category
- Endpoint: /{id}
method: PUT
description: modify category by id
- Endpoint: /{id}
method: DELETE
description: delete category by id

Recipes: /recipes/

- Endpoint: /
method: GET
description: retrieve list of recipes
- Endpoint: /
method: POST
description: create a new recipe
- Endpoint: /{id}
method: PUT
description: modify recipe by id (shadow)
- Endpoint: /{id}
method: DELETE
description: delete recipe by id (shadow)

Dishes: /dishes/

- Endpoint: /
method: GET
description: retrieve list of dishes
- Endpoint: /
method: POST
description: create a new dish/dishes
- Endpoint: /{id}/action/{type}
method: PUT
description: modify status of dish (cooking)
- Endpoint: /{id}
method: PUT
description: modify dish by id
- Endpoint: /{id}
method: DELETE
description: delete dish by id



```
OrderArchives: /order_archives/
- Endpoint: /
  method: GET
  description: retrieve list of archived orders
- Endpoint: /{id}
  method: POST
  description: archive an order
- Endpoint: /{id}
  method: PUT
  description: modify archived order by id
- Endpoint: /{id}
  method: DELETE
  description: delete archived order by id
```

5.4. Access to routes

As stated previously during the discussion on database design, the permissions to access certain collections will be handled by the backend. To achieve this, the web application employs JSON Web Tokens (JWTs), storing user-related information and account type within the token. Below is an example 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": 1704798129,
  "exp": 1704970929
}
```

When a client accesses a route and the token has been verified, the application checks whether the user has access to the specific route using this code snippet:

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

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

This code segment ensures that only users with the “admin” role are granted access to the specific route.

Here are the tables displaying all the routes along with the roles that have permissions to access them.



Route	Method	Permissions	Route	Method	Permissions
/categories	GET	ALL	/orders	GET	WT, PD, CS
/categories	POST	AD	/orders	POST	WT
/categories/{di}	PUT	AD	/orders/{id}/action/{choice}	PUT	WT, PD
/categories/{id}	DELETE	AD	/orders/{id}	DELETE	CS
/dishes	GET	WT, PD, CS	/recipes	GET	ALL
/dishes	POST	WT	/recipes	POST	AD
/dishes/{id}/action/{type}	PUT	PD	/recipes/{id}	PUT	AD
/dishes/{id}	PUT	WT	/recipes/{id}	DELETE	AD
/dishes/{id}	DELETE	CS	/restaurant_informations	GET	ALL
/ingredients	GET	ALL	/restaurant_informations	POST	AD
/ingredients	POST	AD	/restaurant_informations/{id}	PUT	AD
/ingredients/{id}	PUT	AD	/rooms	GET	ALL
/ingredients/{id}	DELETE	AD	/rooms	POST	AD
/order_archives	GET	AN	/rooms/{id}	PUT	AD
/order_archives/{id}	POST	CS	/rooms/{id}	DELETE	AD
/order_archives/{id}	PUT	X	/tables	GET	ALL
/order_archives/{id}	DELETE	AN	/tables	POST	AD
/user/login	POST	ALL	/tables/{id}/status/{type}	PUT	WT, CS
/user	GET	AD	/tables/{id}	PUT	AD
/user	POST	AD	/tables/{id}	DELETE	AD
/user/{username}	PUT	AD			
/user/{username}/	DELETE	AD	/utility/resetCache		AD(used for debug)

Legend:

Admin: AD

Production: PD

Cashier: CS

Analytics: AN

Waiter: WT

Everybody: ALL



5.5. Authentication

In this chapter, let's clarify how authentication using JSON Web Token is managed.

All authentication-related operations are housed within the "auth.middleware.ts" file. Within this file, the token interface is declared. Here is the snippet:

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

Additionally, this file contains several functions such as `authenticate` (basic strategy) for validating the account, `create_token` responsible for token generation, `authorize` to check the validity of a token, `blacklistUser` used for blacklisting a token, and `checkBlacklist` that verifies if the token exists in the Redis blacklist (see next chapter). These methods are directly called by the routes.

Note: all the configurations related to the JWT are being stored in the `.env` file and are being accessed through the "app.config.ts" file. Here are the information related to the token in the `.env` file:

```
#TOKEN SECTION
JWT_SECRET = "SUPER_DUPER_SECRET_PASSWORD"
  #Time expiration token (in hours) when creating
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'
```



5.6. Redis

The backend optimizes performance by utilizing Redis, an in-memory data structure. Redis serves as a cache for certain collections and manages token blacklisting using the TTL (time to live) function. Access to Redis is facilitated via the "redis.service.ts" file, which encompasses a generic class designed to store diverse information in a key-value structure.

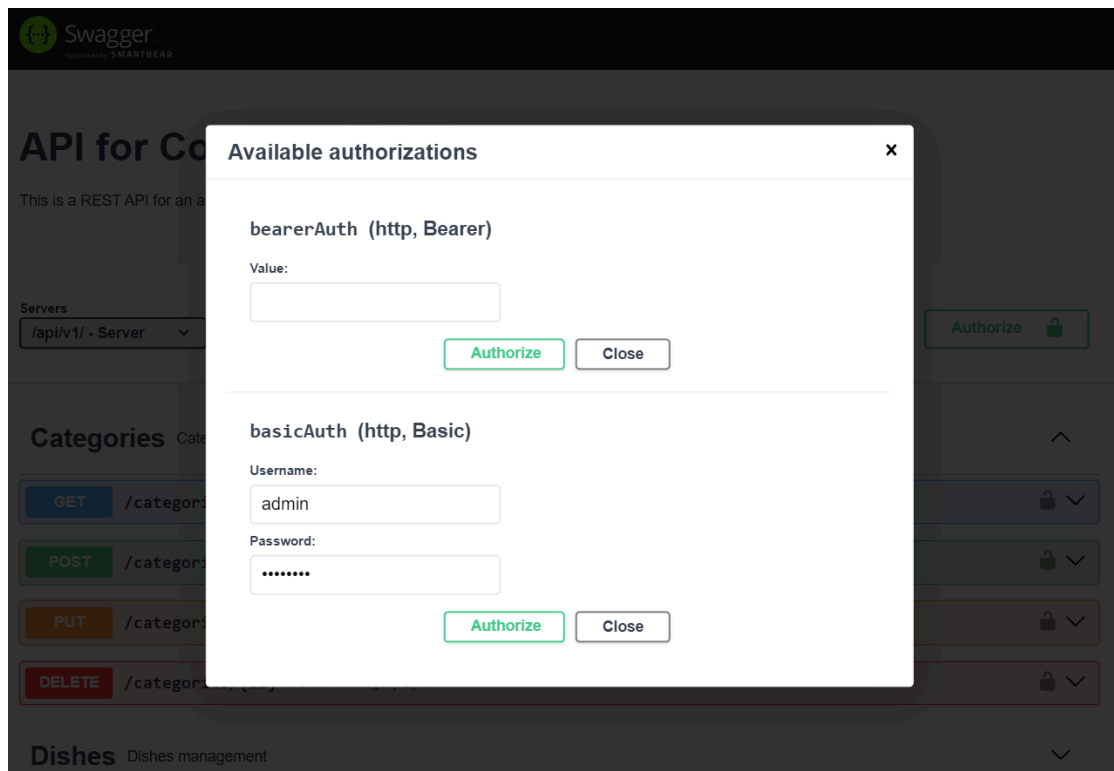
Not all collections are stored in the cache; collections that undergo frequent editing are excluded. Moreover, if cached collections are updated, the cache is cleared and reset. The following collections are cached: *Categories*, *Recipes*, *Ingredients*, *Restaurant Information*, and *Rooms*.

The Redis cache stores all of the backlisted tokens, which are marked as such only when an administrator modifies a user. To maintain data integrity, tokens preceding the modifications are blacklisted in the cache until they expire.

Note: if the cache server is down the application stops working and returns to all the requests an error message that the Redis service is down.

5.7. Swagger

Swagger is employed by the team to comprehensively document and test routes and input structures used in queries. Accessing Swagger is simple, just use the URL "<http://localhost/api/v1/docs/>". To access the routes, authentication is required. Click the "Authorize" button, which prompts a window (as shown in the image below) where you can input your credentials using basic authentication. Afterward, utilize the "/user/login" route to obtain the token, which should be pasted into the bearerAuth section. Then you can test all the routes.





5.8. Socket.IO

In order to ensure that the clients have information that is up to date the Socket.io library is integrated to update the clients when there are changes in the records of a certain collection.

For that, clients when exposed to the data in a specific component subscribe to a socket service and when there are updates to the backend, it sends a broadcast signal with a specific collection channel. With the following snippet:

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

After the clients receive the signal the request updates using the normal API routes.(for the frontend side see chapter 6.33)

6. Frontend

The Angular frontend, driven by the Angular framework and styled using CSS, establishes a structured architecture for intuitive development and seamless user interface design in a single-page application format. Integrated with the Mat Angular component library, it enriches the application with rich and responsive graphical elements, ensuring a cohesive and visually appealing user experience while maintaining scalability and responsiveness.

6.1. Structure of the Application

As in the backend organization of the application is fundamental for a good development. For this we decided to adopt the LIFT structure guidelines:

- **L**ocate code
- **I**dentify the code at a glance
- Keep the **F**latteest structure you can
- And **T**ry to be dry

For this it is fundamental the use of modules to help group and encapsulate code in a consistent structure. These will be the types of modules used to organize the application:

- **Root Module:** it just takes responsibility loading all the other modules (a.k.a. app module)
- **Core Module:** provides all the essential elements like components, services, etc... to the whole application
- **Feature Modules:** each module contains features relative to a single aspect of the application, in case of the CookHub application each module contains all the elements related to a role.(modules folder)

So analyzing the workflow of the modules its goes in this order
RootModule->CoreModule->FeatureModule

Below is displayed a simplified version of the Angular frontend files:



```
.
├── 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 Module

As stated before the Core component takes care of all the essential functionalities.

Inside the master module is managed the main routing by importing all the feature routers as children, here is an example:

```
{
  path: 'admin',
  loadChildren: () => import('../modules/admin/admin.module').then(m => m.AdminModule),
  canActivate: [authGuard],
  data: { type: 'admin' }
},
```

The module includes some basic components like:

- *master-container*: used for the toolbar and sidebar that persist through the whole application except auth page
- *logo*: a dummy container showing showing the first page
- *notifier*: component that its called when the app want to shows messages of confirmation from backend
- *error-page*: shown when you encounter errors like invalid url etc...

6.2.1. Models

The core model also contains the models that are simple interfaces of the database collections(they are almost identical to the interfaces used by the backend) and other elements used by the application

6.2.2. Guards

This module includes a single guard, "auth.guard.ts", used for module routing access. It checks the presence of a valid token(see "auth.service.ts" in the next section) and verifies the user's role within the token to determine access permissions based on the accessed route. Here is a snippet of how it checks:

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

When the user tries to access a resource that he is not allowed, he is redirected to the error page that says this resource is forbidden.



6.2.3. Services

The core also contains all the services used by the application:


- **api.service.ts**: used to make requests to the backend and handle the responses.
- **auth.service.ts**: used for the authentication requests token and extracts it into the memory.
- **database-references.service.ts**: this service stores inside the application some lightweight collections like ingredients, categories, recipes... so the client doesn't make a request every time it needs them, improving the workload of the backend server; those references are only initialized at the start of the application(after login) and then updated only if those collections have changes in them through the socket service.
- **page-data.service.ts**: service used to pass generic data between one component to the other, used by waiters for example to transfer order data before pushing it to the database.
- **page-info.service.ts**: simple service used to store the name of the page that is displayed at the center of the topbar-.
- **socket.service.ts**: a simple service that listens and emits signals using the socket.IO library.

6.3. Feature Modules


In this section, will be explained each of the modules, what they do, some highlights worth mentioning and images for demonstration.

6.3.1. Auth

Simple module that contains only the login page, after getting access it redirects the user to the core model homepage.

Login 

Username*
admin

Password*
..... 

☒ Remember me

Login



6.3.2. Admin

In this module are present all the pages that allow the administrator/owner of the restaurant to make changes to the data in the database: *Users, Categories, Recipes, Ingredients, Information about restaurant, Rooms and Tables*. And it archives that with the use of dynamic components in order to avoid repetition. So for this scope the three components were developed: **dynamic_form** to create new records, **dynamic_table** to display the existing records, and **dynamic_table_form** to edit a specific record from the table.

How do these dynamic components work? In the parent component you can find the **models** for the construction of the dynamic components. Those models follow the “iDynamicForm, iDynamicTable, iDynamicTableForm” interfaces that can be found in the core models. Those *object models* are defined and specify the route, columns, text fields, etc... that should be displayed by the dynamic components. The dynamic components receive the input object and display the corresponding elements.

```
modelInput: iDynamicForm = {
  route: '/categories',
  formName: 'newCategory',
  textFields: [
    {
      name: 'name',
      label: 'Name',
      type: 'text',
      required: true,
      value: '',
    },
    {
      name: 'color',
      label: 'Color',
      type: 'text',
      required: true,
      value: '',
    },
    {
      name: 'order',
      label: 'Order',
      type: 'number',
      required: false,
      value: '',
    }
  ],
};
```

dynamic_form:

This is the object used to build the form of categories.

Inside the objects are displayed all the fields or other elements that the user will have to compile in the form and other information.

- *route*: to which the data will be send
- *formName*: variable name for the form
- *textFields*: defines a text field that will be displayed by the dynamic component
- *checkboxes*: simple true/false checkboxes
- *arrayTextFields*: text fields that are inserted into an array object
- *elementsFromDatabaseSingleChoice*: generates a mat selector which option are extracted from the database(can select single element)
- *elementsFromDatabaseMultipleChoice*: same as the previous one but the user can choose multiple elements



```
modelTable: iDynamicTable = {
  route: '/categories/',
  archive: false,
  tableListener:
    eListenChannels.categories,
  columns: [
    {
      name: 'name',
      label: 'Name',
      type: 'text',
    },
    {
      name: 'color',
      label: 'Color',
      type: 'text',
    },
    {
      name: 'order',
      label: 'Order',
      type: 'text',
    },
  ],
  expandable: false,
  subModelInput: {
    ...this.modelInput as
    Partial<iDynamicTableForm>,
    formName: 'modifyUser',
    routeModify: '/categories/',
    routeDelete: '/categories/',
  },
}
```

dynamic_table:

This is the object used to build the table.

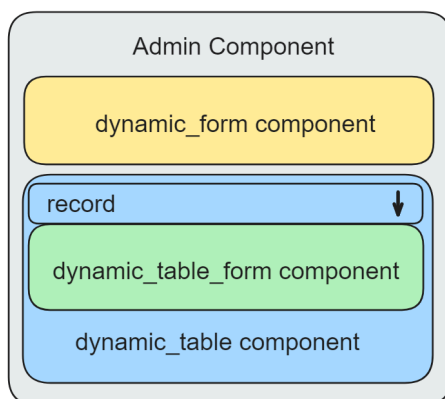
It passes to the component simple elements like the route from where to retrieve the data, the socket channel that it needs to listen to, and columns to be displayed.

If the records can be changed it you pass inside this object the “subModelInput”.

dynamic_table_form:

It uses a similar object to the *dynamic_form* to create a form with precompiled elements for the to be modified.

Note: Due to how the MatAngularTable works if they have the expandable option selected, it tries to emulate all of the expandable records (that would be a problem with lots of records), so to improve performance the dynamic_table component uses an array “selectedRowCheck” to make the MatAngularTable compile only the selected row and pass the correct data to the dynamic_table_form component.



Here is a representation of the parent component, and the dynamic sub-components.

Below will be displayed two photos as example:



Username Name Surname

Phone Password Rooms

Categories

☐ Admin ☐ Waiter ☐ Production ☐ Cashier ☐ Analytics

Submit

Filter

username	name	surname	phone
admin	admin	admin	3506342301
JoeyAdmin	Joey	Tribbiani	3506342302
RachelWaiter	Rachel	Green	3506342303
GuntherWaiter	Gunther	Unknown	3506342308
JaniceWaiter	Janice	Hosenstein	3506342309

Items per page: 5 1 - 5 of 12

Filter

username	name	surname	phone
admin	admin	admin	3506342301
JoeyAdmin	Joey	Tribbiani	3506342302

Username Name Surname
JoeyAdmin Joey Tribbiani

Phone Password Rooms
3506342302

Categories

☒ admin ☐ waiter ☐ production ☐ cashier ☐ analytics

Submit Delete

RachelWaiter	Rachel	Green	3506342303
GuntherWaiter	Gunther	Unknown	3506342308
JaniceWaiter	Janice	Hosenstein	3506342309

Items per page: 5 1 - 5 of 12

6.3.3. Waiter

In the waiter module are grouped all pages used by the waiter to take orders, and marked served courses of orders. So let's discuss the two parts:

- Taking orders: this part is handled by the **orders-table** and **order-form** components. Where the waiter inputs the number of customers, room and tables and confirms it, and the order is added to the table. Then the waiter can select the specific order to take information from the customers. This opens the **order-detail** component where he will see the details of a specific order, the waiter can choose to add courses and dishes to the order, this will open the **menu-selector** component that will list all the elements from the menu. After the waiter selects all the dishes and courses he can confirm the order in the **order-detail** and all the choices selected will be pushed to the database.

Below will be displayed the images that the waiter will see:



New Order

Guests* 5 Room 2 Table Table 21, Table 22

Submit

Filter

Rooms	Tables	Guests	Status
1	Table 11	10	ordering
1	Table 12	10	delivered

Items per page: 10 1 - 2 of 2

Order detail of tables:

Table 21,Table 22

Detail Order:

Room: 2
Table: Table 21,Table 22
Guests: 5
Capacity: 20
Status: waiting

About to be added Courses:

Add Course

Submit Order

Adding a new Dish to the course:

Drinks Appetizers First Dishes Second Dishes Dessert

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 14€

Send

Order detail of tables:

Table 21,Table 22

Detail Order:

Room: 2
Table: Table 21,Table 22
Guests: 5
Capacity: 20
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: 14€

Notes:

Dish: Espresso Martini Vodka, coffee liqueur, espresso, and a splash of simple syrup.

Price: 12€

Notes:

Dish: Berry Blast Smoothie Blend of mixed berries, yogurt, honey, and a splash of orange juice.

Price: 9€

Notes:

Dish: Stuffed Mushrooms Mushrooms filled with a blend of herbed breadcrumbs, parmesan, and cream cheese.

Price: 9€

Notes:

Dish: Spinach and Artichoke Dip Creamy blend of spinach, artichokes, and melted cheese, served with tortilla chips.

Price: 12€

Notes:

Add Course

Submit Order



-Serving orders: first we need to clarify that the application operates in courses(in italian "portate") so after a course is prepared by the kitchen/bar a waiter is notified to serve it, and after all the courses of an order are served the order is completed.

So when the course is marked completed by a production account, all the waiters receive a notification(through the notifier core component). And the above right notification counter is increased by one.

The waiters can navigate to the **ready** component where they can confirm a course that they have served.

Ready to serve courses

Course of Table 21, Table 22 Number of dishes: 6

Dishes

Status: ready Dish: Classic Mojito	Started by: admin At: 01:42 Ended by: admin At: 01:42
Status: ready Dish: Strawberry Basil Lemonade	Started by: admin At: 01:42 Ended by: admin At: 01:42
Status: ready Dish: Espresso Martini	Started by: admin At: 01:42 Ended by: admin At: 01:42
Status: ready Dish: Caprese Skewers	Started by: admin At: 01:42 Ended by: admin At: 01:42
Status: ready Dish: Crispy Calamari	Started by: admin At: 01:42 Ended by: admin At: 01:42
Status: ready Dish: Stuffed Mushrooms	Started by: admin At: 01:42 Ended by: admin At: 01:42

Serve Course

6.3.4. Production

The production module is reserved for the kitchen/bar, but because they are literally similar in functionality the team decided to merge them together. So to distinguish them the account can have categories assigned to them, and they can only cook/prepare recipes in that category. So for example the account *PhibyBartender* has the category *drinks* assigned to her, when she sees her queue of preparation she will only be able to prepare the drinks in her category.

The production module has only the **queue** component inside which is a FIFO style queue that operates on courses and it mixes them with the courses of other orders.

There a cook/bartender can select the dishes he/she can start cooking and finish cooking, when all the dishes of a course are completed the cook/bartender can confirm the course. Confirming the course will notify the waiter to serve the course(the section above)



Here is he screenshot of the production queue:

Your Queue Categories are: Appetizers , First Dishes , Second Dishes , Dessert

Course of Table 21, Table 22 Number of dishes: 3

Dishes

Status: waiting **Dish:** Espresso Martini
Status: working **Dish:** Crispy Calamari
Status: waiting **Dish:** Spinach and Artichoke Dip

Started by: admin **At:** 02:01 [Finish Cooking](#)
[Start Cooking](#)

Course of Table 21, Table 22 Number of dishes: 1

Dishes

Status: ready **Dish:** Crispy Calamari

Started by: admin **At:** 02:02 **Ended by:** admin **At:** 02:02

[Confirm Course](#)

6.3.5. Cashier

Cashier Module is responsible for the cashier functions in a restaurant.

So it cashes the orders, archives them in the archive, and displays what tables are free at the moment.

The cashier starts with the **cashout** component that contains a list of active orders, when an order is completed the cashier is able to check the order. This opens a **order-detail** component, a simple recap of the order (a check). There the cashier can finally confirm, and the order is transferred from the order collection to the archive.

Also the cashier has the **tables** component a simple table of tables with their statuses displayed.



Below displayed screenshots of the cashier components

Cashout

Cashout Orders

Order of Table 11

Number of courses: 1

Number of people: 10

Final Charge: 51€

General Info of Orders:

Guests: 10 people

Capacity: 10 people

Tables: Table 11

Room: 1

Service charge: 30€

Courses:

Course served by: admin At: 12:03

Course number: 1

Dish: Classic Mojito Price of dish: 13€

Modification: Mozzarella cheese Type: add 3€

Dish: Strawberry Basil Lemonade Price of dish: 8€

Final Price: 51€

Cashout Order

Order of Table 12

Number of courses: 0

Number of people: 10

Final Charge: 30€

Order of Table 21, Table 22

Number of courses: 3

Number of people: 5

Final Charge: 15€

Cashout Detail Order

Cashout Order details

General Info of Restaurant:

Restaurant: CookHub

Address: Galileo

Email: cook@hub.com

Phone: 3456787654

IVA: 3436535634

General Info of Orders:

Guests: 10 people

Charge per Person: 3€

Service charge: 30€

Capacity: 10 people

Tables: Table 11

Room: 1

Courses:

Course number: 1

Dish: Classic Mojito

Price of dish: 13€

Modification: Mozzarella cheese Type: add 3€

Dish: Strawberry Basil Lemonade

Price of dish: 8€

Final Price: 51€

Cashout Order

Cashout

Cashout Detail Order

Status Of Tables

Filter

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

Items per page: 5 1 - 5 of 25



6.3.6. Analytics

Analytics is the role that has access to the archives of the orders and can visualize them and access statistics of them. So the analytics module has two main components:

- **archive** component that can be used to access the history of the orders and see the orders on a certain date/period.
- **statistics** component that displays simple charts (using chartJS) showing the performance of the restaurant, the waiters, and production.

Note: those are simple statistics and not properly optimized for performance, and if they have big data to process, will require a few seconds to load.

Screenshots of Analytics module components:

Date Range Filter

Filter
1/1/2023 – 12/31/2023
DD/MM/YYYY - DD/MM/YYYY

☐ Logs **Apply Filter**

Order: Table 53	Time and Date: 20:01 31-12-2023	▼
Order: Table 51	Time and Date: 12:01 31-12-2023	▼
Order: Table 55	Time and Date: 11:37 31-12-2023	▼
Order: Table 33	Time and Date: 04:57 31-12-2023	▼
Order: Table 45	Time and Date: 15:40 30-12-2023	▼

Items per page: 5 1 – 5 of 896 < >



7. Conclusions

The development of CookHub provided the team with a new understanding of the MEAN technology stack. This was the first encounter with these technologies for all team members, and after an initial learning phase, the application's development proceeded without critical issues.

Although this application is a simple prototype lacking many features necessary in a real life situation, it still aims to cover all of the essential aspects of the application. There are places where it can improve, mainly in the statistics department by incorporating dedicated functions and processes to improve the performance in this aspect

Additionally, the experience of developing the infrastructure to host this application through Docker was excellent. It allowed us to use it in various environments without encountering problems and configure how the components interact with each other easily.

In conclusion the development of this application allowed us to put into practice what was learned during the lessons.