Distributed Cafè System Report

Elisa Albertini
elisa.albertini3@studio.unibo.it

Eleonora Bertoni
eleonora.bertoni3@studio.unibo.it

December 23, 2024

Our project aims to implement a distributed system for a cafè. It involves three front-end applications relying on a microservices-based back-end.
The first application is used by the cafè employees to manage the customers orders, made by the second application, in real time. The last one provides some functionalities to manage the warehouse.

1 Goals of the project

1.1 Client interview:

In the following section it's reported the first interview with the client. *"I'm the owner of a cafe and I'm quite interested in introducing new technologies to help my activity to modernize since my customers and my employees are generally really young. My idea is to have three different software to use: I was thinking about a web application for the customers, that will allow them to make orders, a software for our PC, that the waiters and the kitchen staff can use and a web application for our managers to help them manage the inventory and the menu of the cafe. I had this idea because we deal with everything on paper and it starts to be quite complicated. The menu changes quite frequently because we always come up with new ideas. Also, when an item runs out, my waiters have to remember it because there is no way to indicate it on the old paper menus. In addiction, I want to avoid long queues for take away and too many phone calls to make home delivery orders. All the orders will be made by the application and when they are ready the system will notify the customer using the email he left at the moment of the order. Paper orders go missing quite frequently and sometimes they are made twice. I want to avoid that, too. Also, the inventory is quite complicated to do manually so some kind of database would be fantastic. In the first version of this project there is no need to implement any payment system. I'd like something quite simple and usable, we are not a fancy cafe!".*

1.2 Usage scenarios

1.2.1 Use case of the orders management

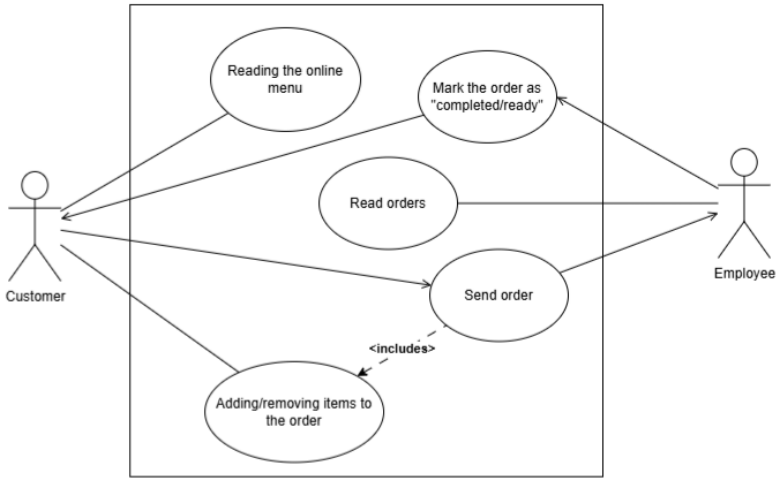

Figure 1: Use case of the orders management

This use case (figure 1) represents the interaction between the employee actor and the customer actor. The first one can read the menu from the web app, must add items to the cart and can send the order. The order is received by the employee. He can read all the pending orders from the PC software and, when the order is completed or ready, he marks it.

1.2.2 Use case of the manager modifying the menu

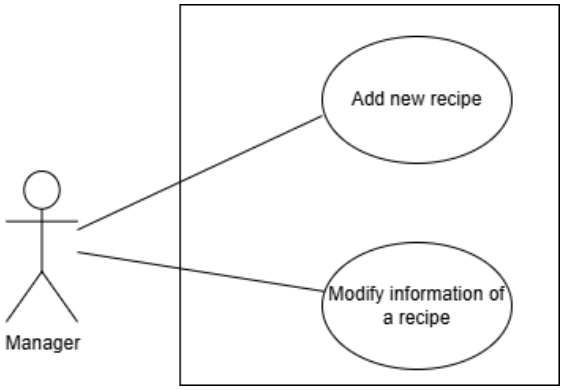

Figure 2: Use case of the manager modifying the menu

This use case (figure 2) represents the management of the menu. The actor is the

manager. He is allowed to do different actions: he can add a new recipe and modify old ones along with their information.

1.2.3 Use case of the manager interacting with the warehouse

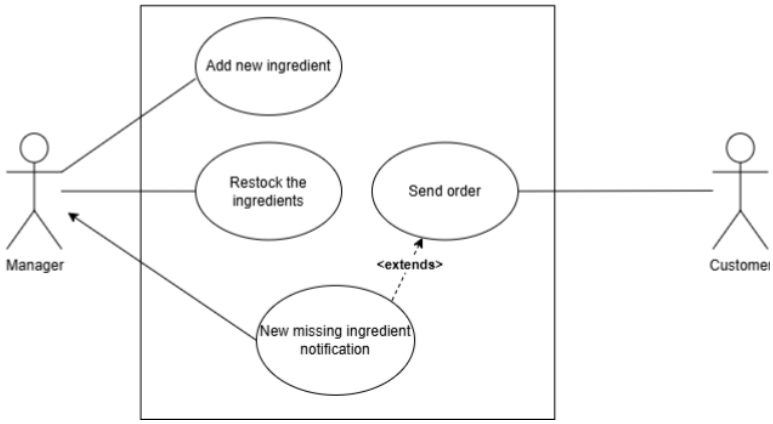

Figure 3: Use case of the manager interacting with the warehouse

This use case (figure 3) represents the management of the warehouse. The actor is the manager. He is allowed to do different actions: he can add a new ingredient and restock it by updating its quantity. In addiction the manager receives a notification if an ingredient runs out after a customer has made a new order.

1.3 Definition of done

The deliverables of the project are:

- Back-end:
  - **WarehouseService**: microservice exposing API for the warehouse management
  - **menu-service**: microservice exposing API for the menu management
  - **orders-service**: microservice exposing API for the orders management
  - **server**: server exploiting web socket connections that interacts behind the scenes with microservices
- Front-end:
  - **EmployeeApplication**: a software application that manages the orders arriving from the clients meant to be used by the employees
  - **manager-application**: a web application that manages the warehouse and the menu meant to be used by the manager

– **customer-application**: a web application that allows the customers to make orders

The *Back-end* components are considered done when all the requirements are met and the tests are implemented and successfully passed.
The *Front-end* components are considered done when all the requirements are met and they are successfully tested manually.

2 Requirements Analysis

2.1 Application requirements:

2.1.1 Customer application

This application allows the customers to:

- read the menu (it shows just the available items)
- add an item in the order recap
- modify the order recap
- choose the type of order (from the table, take-away, home delivery)
- send the order recap
- receive emails from the system

2.1.2 Employee application

This application allows the employees to:

- visualize the recaps of the different orders in real time
- complete an order

2.1.3 Manager application

This application allows the managers to:

- visualize the available ingredients in the warehouse and their quantity
- restock the ingredients the warehouse ran out of
- receive a notification when the warehouse runs out of an ingredient
- visualize the ingredients information
- insert or modify items from the menu and their recipe too
- visualize the items information

2.2 System requirements:

The goals of our project is to:

- Front-end:
  - implement a Java application (for the employees)
  - implement an Angular application (for the customers)
  - implement an Angular application (for the managers)
- Back-end:
  - implement microservices exposing REST API
  - implement a web socket connection with a server that interacts behind the scenes with microservices

2.3 Business requirements:

- Allowing the customers to make orders in an easier way using a web app
- Allowing the employees to manage the orders in real time from an application
- Allowing the management of the menu in an easier way using a web app
- Allowing the management of the inventory in an easier way using a web app

2.4 Requirements list

- Functional
  - Web app for customers:
    - (Menu)
      - list available menu items
      - add an item in the order recap
    - (Orders)
      - modify the order
      - choose the type of order (from the table, take-away, home delivery)
      - send the order
      - receive emails from the system
  - Software for employees:
    - (Orders management)
      - show recap of the different orders in real time
      - change the status of the order

  - Web app for manager:
    - (Warehouse management)
      - manage ingredients and their quantity
      - restock missing ingredients
      - keep some ingredients information
    - (Menu management)
      - manage items and their recipe
      - keep some items information
    - (Notifications)
      - receive missing ingredients notifications

- Non functional
  - Each component of the back-end must:
    - run correctly on Linux
    - support Java Runtime Environment version 17
    - support Node.js version 20

3 Design

In this section we describe the client-server architecture of our Distributed System.
Distributed Cafè System needs to deal with requests coming from three front-ends meant to be used by different users (customers, employees and manager). In order to manage these communications we divided the back-end into a server and three different microservices, one for each relevant portion of the domain (menu, warehouse and orders).
The front-ends interact only with the server whose main task is to sort all the incoming calls to the microservices. For many calls, the server simply waits for the microservice response and sends it back. For those functionalities that are implemented relying on more than one microservice operations, the server needs also to manage the interactions between the microservices: it decides the order of the requests, waits for their responses and, based on the results, manages the next step dealing with errors if needed.

3.1 Interaction

We examine the interactions between the different components using sequence diagrams. We provide an UML schema for all functionalities relying on a complex communication involving multiple microservices in figures 4, 5, 6. We also show different outcomes for the most important conditions, for readability we omit trivial error cases.
In figure 4 we show how the server provides a list of all available menu items, namely, the items made by ingredients that are available in the warehouse. First, it waits for the list of available ingredients from the warehouse service, and then it uses this result

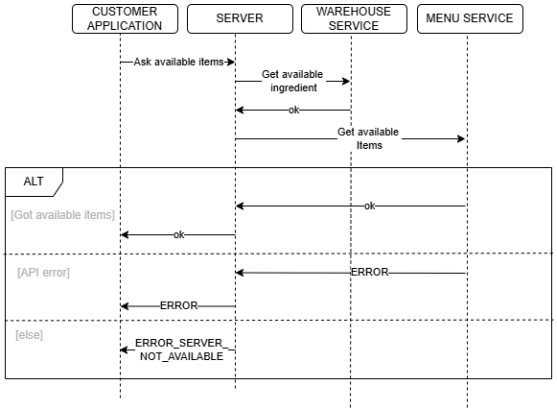

Figure 4: Sequence diagram for "get available items" functionality

as an input to delegate the task to the menu service.
For "put order", in figure 5, the server collects the order that has to be modified and

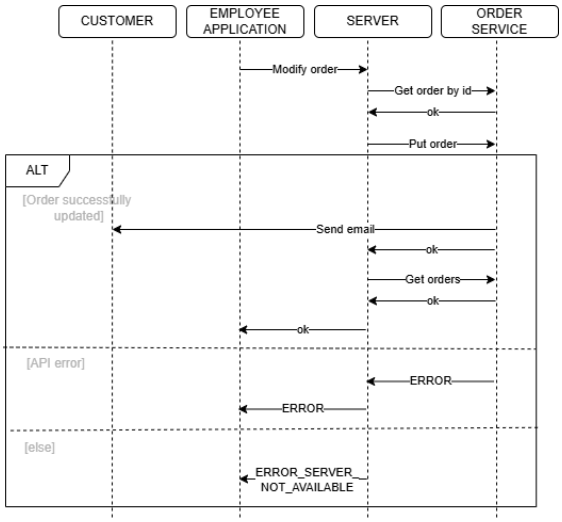

Figure 5: Sequence diagram for "put order" functionality

updates it. After that, it returns all the orders. When a take-away order or a home delivery order becomes ready, it sends an email to the customer.
Figure 6 describes what happens when a new order is sent from the customer. Firstly, the server checks if all the needed ingredients by the item recipe are present in the warehouse asking both warehouse and menu services. If the order cannot be accepted an error is sent to the customer application. In the opposite case, it calls the order service to create the order. After that, it removes all the used ingredients from the warehouse, and in case an ingredient becomes missing, it sends a notification to the manager application. In the end, it sends a notification to the customer application to tell that the order has

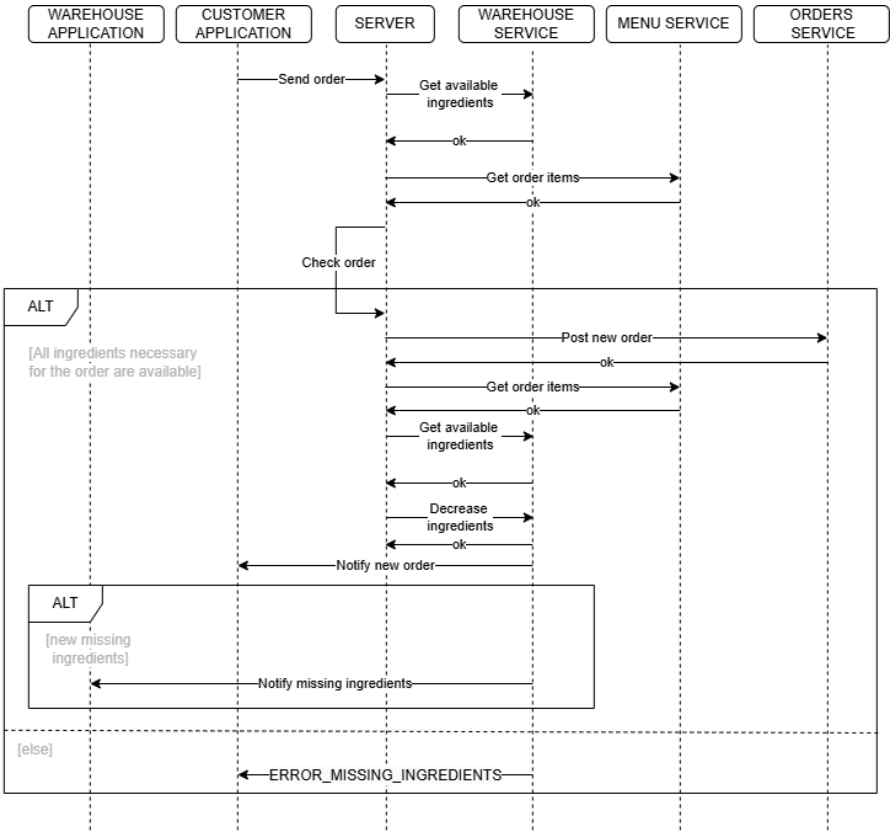

Figure 6: Sequence diagram for “create order” functionality

been successfully created.
In figure 7 we describe "create ingredient" to provide an example for all the interactions

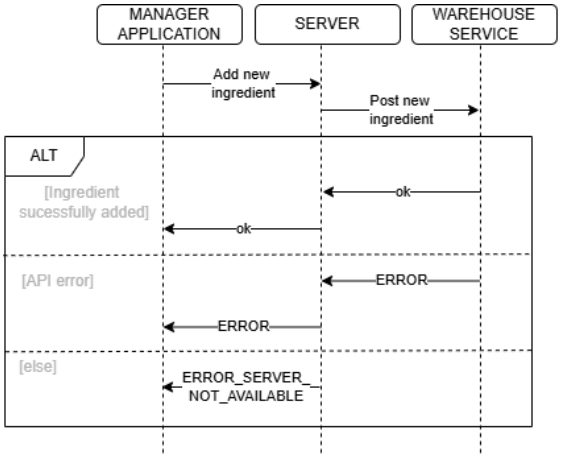

Figure 7: Sequence diagram for "create ingredient" functionality

involving just one microservice functionality. In this case the server calls the microservice and returns the answer, either the result or an error message. We use a specific error if the microservice is not available.

3.2 Architecture

The front-end applications communicate only with the back-end server via web socket while the server communicates with the microservices via REST API (figure 8).

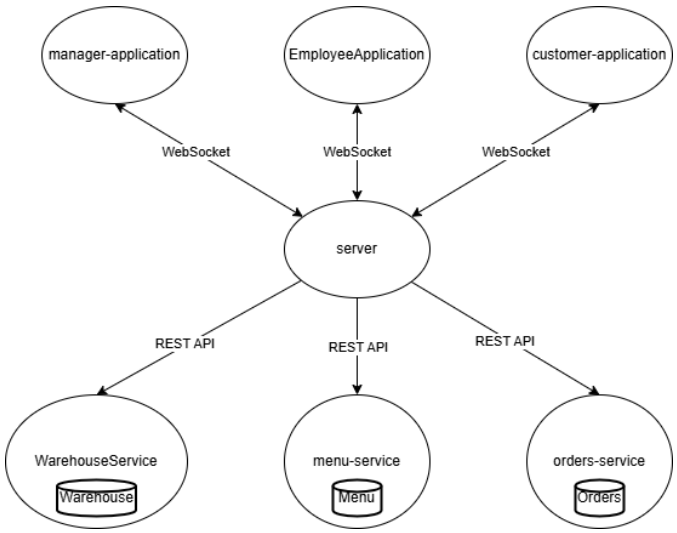

Figure 8: Interaction diagram

3.2.1 REST

The Representational State Transfer (REST) architectural style is based on HTTP protocol.
REST components communicate by transferring a representation of a resource in a format matching one of a set of standard data types. The representation remains hidden behind the component uniform interface. Each resource has to have at least one identifier in the form of a URI and no two resources can be the same, since each resource maps a different concept.

The architecture follows the *client-server pattern* which is composed of a server component that offers a set of services and listens for requests upon those services, and a client component that desires that a service is performed so it sends a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client.
This improves *portability*, *scalability* and *evolvability*.

The request are *stateless* which means that each of them must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.
This property improves *visibility*, *reliability*, *scalability* and *simplicity*.

Another principle applied to REST is the principle of *generality*, which is applied to the component interface. It's a central feature that distinguishes REST from other network-based styles because of its emphasis on a *uniform interface* between components.
This principle improves *simplicity*, *visibility* and *evolvability*.
The drawback of this is that it actually degrades efficiency by transferring information in a standardized form rather than one which is specific to an application's needs.

3.2.2 Front-end

- Web applications
  - Both of them follow Angular component-based framework:
    - *app* contains all the components. There is a folder for each component whose implementation is divided into a CSS, HTML and Typescript files
    - *utils* contains useful reusable functions and *schema* folders with interfaces corresponding to the structure of exchanged JSON data

- *manager-application* (most relevant components):
  - *home* is the main web page, it opens a web socket connection to send a *Log* to the server and to receive notifications. If there are missing ingredients it displays a pop-up listing them. Here there are also the *data-table*s showing the menu and the warehouse content
  - *data-table* is the component that shows the menu/warehouse content. It opens a web socket connection to ask the back-end the data it needs to fill the table. This connection is also used to receive the data or the error messages from the server. If the data cannot be retrieved, an error will be written instead of the table. Thanks to other components used by the data-table, it is possible to open a form-like dialog with a button that allows the manager to send update requests through the same web socket. The manager is informed about the failure of the operation by a dialog displaying an error message
- *customer-application* (most relevant components):
  - *menu* displays the available items, requested to the back-end, as *item-card* components. This component opens the web socket connection to receive the menu item data or an error if the retrieval is not possible (there are no available items or the back-end is not reachable). The menu allows to add items to the cart
  - *cart* is the web page that shows a recap of the menu items selected by the customer through the app. It allows the customer to modify the ordered items or to confirm the order after inserting all the requested information. If some input is missing or is wrong the order cannot be sent and a message appears. Here we open a web socket connection that will be used later to tell the customer if the order has been successfully created, emptying the cart, or if something went wrong. Once the customer clicks the button, the request to create the order is forwarded to the back-end through the same web socket
  - *cart-storage.ts* in *utils* implements the access to the local storage used for the cart

- *EmployeeApplication*
  - *WebsocketConnection.java* is responsible of creating the web socket connection to the back-end. When the application starts, it sends the first log message and also the first request to the server asking for all the orders. It also initializes the web socket behavior when a message is received: it requests all the orders everytime a notification arrives and updates the *view* when the server sends the order data. It also tries to reconnect with polling if the server cannot fulfill a request or if the web socket connection is lost.
  - *View.java* and *OrderCard.java* take care of the graphics and repaint feature

- *schema* contains the interfaces representing the order, request and response JSON format

3.2.3 Back-end

- Microservices architecture is structured in layers, for *WarehouseService*, *orders-service* and *menu-service* we have:
  - *domain*: implementation of the domain entities
  - *repository*: encapsulation of the access and management of its own database
  - *application*: encapsulation of the business logic
  - *handler*: management of http requests
  - API: exposed by *routes* in *menu-service* and *orders-service*, exposed by *server* in *WarehouseService*

- *server*:
  - *server.ts* creates the web socket server that listens for new connections from front-ends. The server works with two types of incoming messages formalized in JSON format:
    - *Log* is the first message sent by the front-ends that wish to be notified when a certain event occurs. The server uses it to save the web sockets into two different arrays, one for the *EmployeeApplication*, interested in knowing when a new order is created, and one for the *manager-application*, concerned about the *MissingIngredientNotification*
    - *RequestMessage* is the type of message used by every front-end when they need to use a functionality of the back-end
  - *check-service.ts* is in charge of mapping the incoming *RequestMessage* to API calls and following *handlers*
  - *handlers.ts* in *utils* manages all the calls to microservices API. An handler implements the callbacks deciding the next API calls based on the success or failures of the previous ones. It tackles minor elaborations needed, for example, to use some results as input for certain APIs.
    It also deals with any inactive microservices wrapping exceptions and communicating to the front-end that the entire operation has failed. In the end, the handler always writes the answer on the web socket of the front-end who made the request. In some cases, it notifies other front-ends through the memorized web socket.
  - *schema* contains the JSON structure of the data and the messages the server needs to work with.

3.3 Corner cases

Each microservice manages all the errors related to its database and to the business logic it encapsulates. It also makes sure to tell when it is not possible to elaborate a request due to wrong inputs: the format could differ from the one required by the API or some values could be incorrect for domain constraints.

- These messages are used by the warehouse microservice to signal errors occurred at the database level
  - *ERROR_INGREDIENT_NOT_FOUND*
  - *ERROR_INGREDIENT_ALREADY_EXISTS*
  - *ERROR_EMPTY_WAREHOUSE*
- This message is used when the warehouse microservice cannot complete an operation due to the stored quantity not being enough
  - *ERROR_INGREDIENT_QUANTITY*
- These messages are used by the menu microservice to signal errors occurred at the database level
  - *ERROR_ITEM_NOT_FOUND*
  - *ERROR_ITEM_ALREADY_EXISTS*
  - *EMPTY_MENU_DB*
- These message are used by the orders microservice to signal errors occurred at the database level
  - *EMPTY_ORDERS_DB*
  - *ORDER_ID_NOT_FOUND*
- This message is used by the orders microservice to tell that the requested change is not possible, this can be related to the order current state or type
  - *CHANGE_STATE_NOT_VALID*
- This message is used by the three microservices when the input provided to the API is wrong. This can happen because it does not follow the format expected by the API or because a value is not acceptable
  - *ERROR_WRONG_PARAMETERS*
- This message is the error sent by the server to the front-end when it is not possible to fulfill a request due to one of the microservices involved being unavailable
  - *ERROR_SERVER_NOT_AVAILABLE*

The server handles the microservices errors everytime an API is called, completing the operation with an error as soon as one of the involved API fails or cannot be reached. Front-ends will use the answers given by the server to update the displayed information or to show error messages to the users. When the server is not available the applications show an error message to the users.

4 Salient implementation details

In this chapter, we describe the technologies used to implement our system and we provide the software documentation produced during the development.

4.1 Express

Express [1] is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It provides useful HTTP utility methods and middleware for creating a robust API quickly and easily. It also offers routing functionalities: these methods specify a callback function called when the application receives a request to the specified route and HTTP method. This means that the application "listens" for requests that match the specified route and method, and when it detects a match, it calls the specified callback function.
We used this framework to develop server, menu and orders microservices because we need servers up and running listening to incoming requests.

4.2 Axios

Axios [2] is a simple promise based HTTP client for the browser and Node.js. Axios provides a simple to use library in a small package with a very extensible interface.
We used this package for the server implementation because it needs a way to use the microservices API and manage the calls with promises.
We need Axios also in the tests of *orders-service* and *menu-service*.

4.3 Vert.x

Vert.x [3] toolkit offers many modules to develop reactive applications and web applications both on client and server sides on the JVM. It allows to handle more requests with less resources compared to traditional stacks and frameworks based on blocking I/O. In addition Vert.x is a great fit for all kinds of execution environments, including constrained environments like virtual machines and containers. It is possible to choose a model for asynchronous programming between callbacks, promises, futures, reactive extensions, and *Kotlin Coroutines*. It is flexible: it is not a framework so it does not force a particular structure on the applications allowing the developers to choose what they need from its many libraries.
For the server side, we exploit those functionalities that allow to manage HTTP requests and create API REST following asynchronous programming paradigm. In particular, we

use Vert.x *CoroutineVerticle* and *Kotlin Coroutines* to implement *WarehouseService* and manage the routes handlers. Vert.x also provide a way to create the server HTTP and expose the routes. We also work with *suspend* methods whose execution can be suspended and resumed at any time favoring asynchronous programming mechanisms.
For *EmployeeApplication* we use Vert.x *AbstractVerticle* and *WebSocketClient* which is an asynchronous *WebSocket* client that allows to open WebSockets to servers. We use it to connect to the server and to set the handler when it receives a message from it and to reconnect if the connection is closed. It also allows to write back to the server.

4.4 MongoDB

MongoDB [4] is the most famous document database, designed for ease of application development and scaling. It is a NoSQL database, this means it does not rely on stiff and table-like data structures and prefers flexible documents that match directly with data structures in the code. A document is a data structure composed of field and value pairs, similar to JSON objects. It incorporates related data in one single document to boost performances and minimize computational costs.
We use a MongoDB database for all the microservices. MongoDB provides support for different platforms and languages. In our project this feature allows to use MongoDB even if the microservices work on different platforms, the main difference is the MongoDB client implementation we exploit to connect to the database and query it. We use the *MongoClient* from the *mongodb* npm package for the Node.js services and the one from *mongodb-driver-kotlin-coroutine* library for the kotlin service.

4.5 Angular

Angular [5] is a web framework meant for developers that need to build fast, reliable applications that scale with both the size of the team and the codebase. It is maintained by a dedicated team at Google and provides a broad suite of tools, APIs, and libraries to simplify and streamline the development workflow. The framework guides the developers into writing organized code thanks to its component model and flexible dependency injection system: Angular components make it easy to split the code into well-encapsulated parts while the dependency injection helps you keep your code modular, loosely-coupled, and testable. Angular is open source with an active community.
In our project we follow Angular component-based architecture to write reusable and modular interfaces. Each component encapsulates its own HTML, CSS and Typescript making it possible to manage each part of the application independently. The Typescript file contains a class that defines the operating logic, the HTML file specifies the template of what to display on the page and the CSS file describes the specific style for that component. There is also a selector that defines the name used to include the component in other ones. We used this framework to develop *manager-application* and *customer-application*.

4.6 Web sockets

Web socket is a computer communications protocol, providing a simultaneous two-way communication channel over a single TCP connection.
We need web sockets between the server and the front-ends. The connection is bidirectional: the front-ends use the channel to ask the server for what they need while the server returns the answer. In addition, the server can send notifications through it. Each server functionality can be called sending a specific message, along with the appropriate input, via the web socket. As said previously in 4.3, we use the Vert.x implementation for *EmployeeApplication* while *customer-application* and *manager-application* rely on the Angular web sockets and *server* exploits the *ws* npm package.

4.7 Typia

Typia [6] is a library converting TypeScript types to runtime function.
It is used to ensure type safety in applications, leveraging TypeScript static typing while also providing runtime validation. Instead of defining additional schemas, it's possible to utilize the pure TypeScript type itself.
We used Typia mostly to check if the input data passed to the REST API and the messages sent to the server by web socket are of the correct type.

4.8 Docker

Docker [7] helps developers build, share, run, and verify applications anywhere without tedious environment configuration or management.
We used Docker to build images for the components of our system. For simplification, we do not provide one for the Java app EmployeeApplication, we give an executable jar instead. Here we describe how we build each image as specified in each Dockerfile.

- *WarehouseService*: we chose to start from an image with JDK 17 for the microservice relying on the JVM to run. We provide an environment variable with the address of the database container and then we expose the port. Lastly, we run the shadowJar of the microservice.

- *Node.js back-end components*: we chose to start from an image with Node.js 20, we prefer a light version, for the back-end components relying on Node.js to run. We have two stages using the same image: the first one builds the production version of the component using webpack, which is a npm package for transforming, bundling, or packaging resources and assets, while the second provides an environment variable with the address of the database container and exposes the container port. Lastly, it copies the built version of the project from the previous stage and runs it thanks to Node.js. As for the server component, in the start stage we provide also, as environment variables, the names of the containers of the microservices.

- *Angular applications*: we have two stages: the first one starts from a lighter image with Node.js 20 and build the production version of the application. The second stage starts from the latest image of nginx. Lastly, it copies the built version of the project from the previous stage and exposes the container port.

We rely on Docker Compose to run the containers. We need a *docker-compose* file meant to be used when calling the command *docker-compose up*. We do not build the images here because that step is done in Continuos Integration (CI) in order to publish them to Docker Hub. The Docker Compose has just to pull them.
In the docker-compose file we have the following services:

- mongo
- server
- menu-service
- orders-service
- warehouse-service
- manager-application
- customer-application

Each service has its local port mapped and it connects to a bridge network called DistributedCafeNetwork. Also, some of them have a list of the services they depend on and one or more volumes, depending on the starting image they use.
The services dependencies are the following:

- **mongo**: it has no dependencies since it's the first image that has to be built
- **server**: it depends on warehouse-service container, menu-service container and order-service container
- **warehouse-service**, **menu-service** and **order-service**: they depend on mongo container
- **manager-application** and **customer-application**: they depend on server container

4.9 Produced Documentation

During the development we produced both code documentation and API documentation, fundamental to formalize how the component should interact.
We used Swagger to publish both the web socket documentation, required for the app development, and the API documentation, required for server development. This way we can easily consult them.

For the API documentation:

- **REST API documentation** can be found here
- **Web socket documentation** related to the messages exchanged can be found here

For the code documentation:

- **Back-end documentation**
  - Warehouse service documentation can be found here
  - Menu service documentation can be found here
  - Orders service documentation can be found here
  - Server documentation can be found here
- **Front-end documentation**
  - Manager application documentation can be found here
  - Customer application documentation can be found here
  - Employee application documentation can be found here

5 Validation

We provide tests for the back-end components in order to validate the correct functioning of our system.
Concerning *WarehouseService*, the structure of the test follows the structure of the microservice: we have **JUnit tests** for *repository*, *application* and for the routes in *server*. We use **Jest** to test *repository* and *routes* in *orders-service*, *routes* in *menu-service* and *check-service.ts* and *server.ts* APIs in *server*.
We focus on testing as many functionalities, error cases and conditions as possible to increase the coverage. The routes tests are meant to make a request, wait for the answer and compare it with the expected result. For *server*, we also check the correct delivery of notifications. In this way we test the back-end functionalities derived from the requirements because the tests ensure the correct functioning of the single microservices and also of the server that interacts with them.
Each back-end component has a workflow to run the tests in CI that creates the MongoDB database using a github action. They are used inside the main pipeline for pull requests to check if the tests are successful.

We performed manual tests for the front-ends and we verified that the requirements are met.

If needed, it is possible to run the automatic tests, meant for the CI, locally. You will need Node.js 20, Java 17 and a bash to run:

- Microservices:

  - For *WarehouseService*:
    * ./gradlew :WarehouseService:test
  - For *orders-service*
    * cd orders-service
    * npm install
    * npm test
  - For *menu-service*
    * cd menu-service
    * npm install
    * npm test

- server:
  - First, you need to start each component in a different bash:
    * *orders-service*
      · cd orders-service
      · npm install
      · npm start
    * *menu-service*
      · cd menu-service
      · npm install
      · npm start
    * *server*
      · cd server
      · npm install
      · npm start
    * *WarehouseService*
      · ./gradlew :WarehouseService:run
  - Now you can run the tests:
    * cd server
    * npm test

6 Deployment Instructions

For the correct deployment of our solution, you need Docker Desktop and Java 17 installed in your machine.
After you clone our repository, you can launch our system running the *start.sh* script on a UNIX terminal. The script generates the shadowJar with Gradle of EmployeeApplication and starts it. It also runs the docker-compose file that pulls the docker images from docker hub and starts the containers.
The web applications can be accessed through localhost and the specified port with a browser.

7 Usage Examples

In this chapter we show how to use the three applications.
The examples represent the first time a user interacts with the system after the first execution of *start.sh*, this means that all the databases are empty.
Also, the web applications don't refresh by themselves so the user has to do it instead.

7.1 manager-application

At the beginning the application will show an error message since the database are empty (figures 9).
It is possible to add a new ingredient using the center button in the warehouse page. It will open a form where the user can insert the information required (figure 10). If the ingredient was already present in the warehouse, an error dialog will be displayed (figure 11).
To add a new item in the menu database it works the same way (figure 12 and 13).

All the ingredients stored in the warehouse and the items stored in the menu database are displayed in a table in their own page (figure 14).

In order to restock an ingredient, the user has to click on the "pencil" button on the same row of the ingredient itself. It will open a form that needs to be filled (figure 15). If the restock quantity is illegal, an error dialog will be displayed (figure 16).
In order to update an item (its recipe and/or price), it works the same way (figure 17). In this case is important to select all the recipe ingredient even if it has not to be modified, with the relative quantities. Also the price must be correctly inserted.

In case one of the microservices or the server is not available, a message will be shown to the user (figure 18).

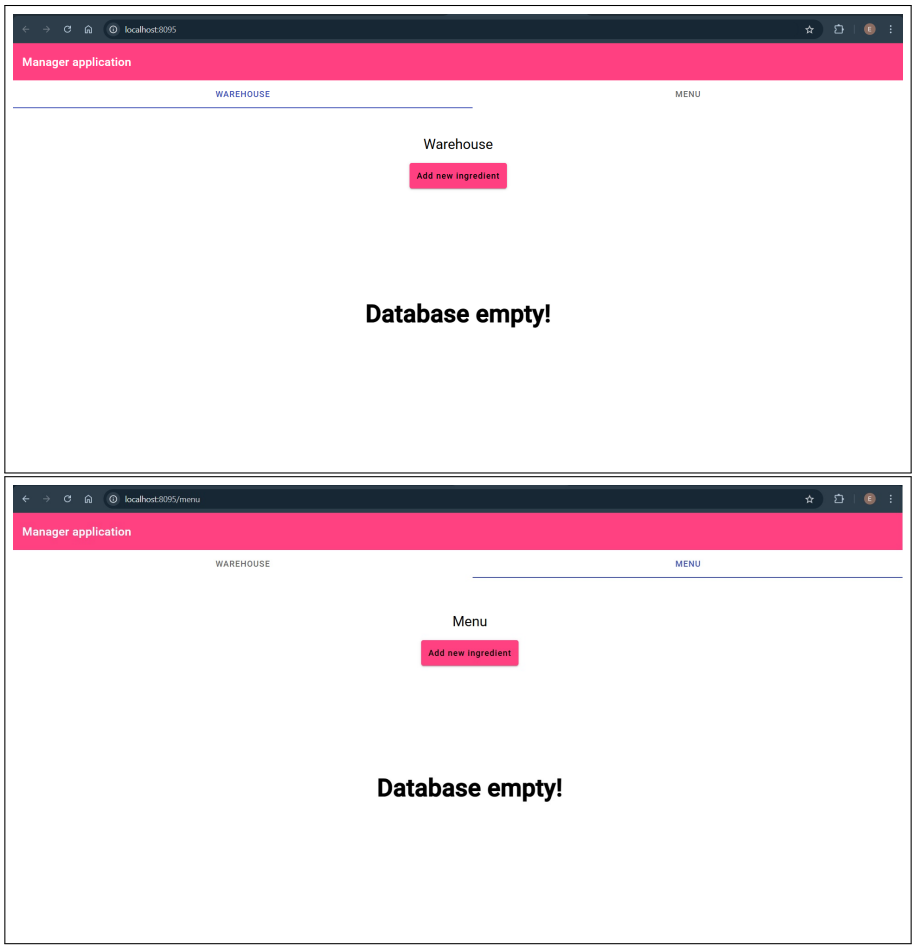

Figure 9: manager-application when warehouse page or menu page shows error empty database

Figure 10: Form used to add a new ingredient

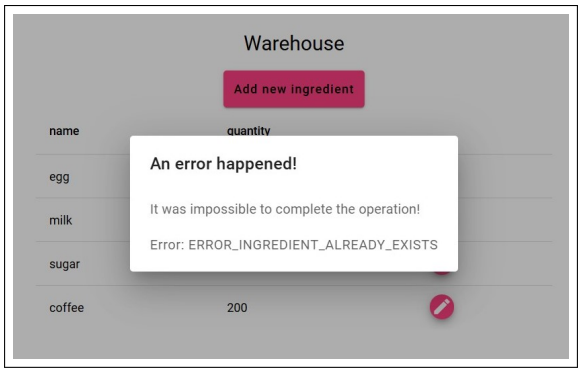

Figure 11: Error dialog

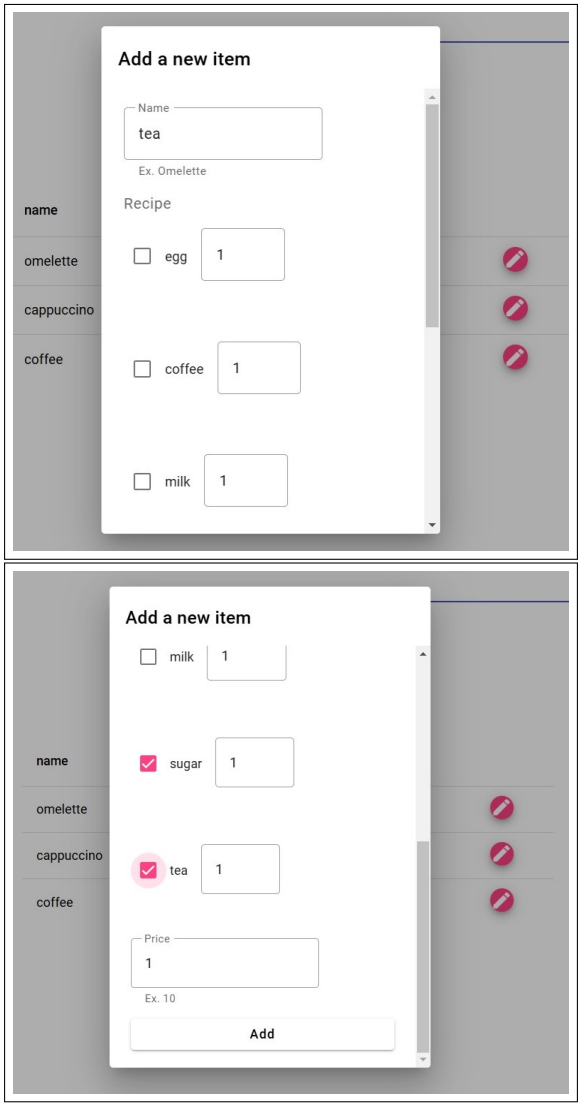

Figure 12: Form used to add a new item

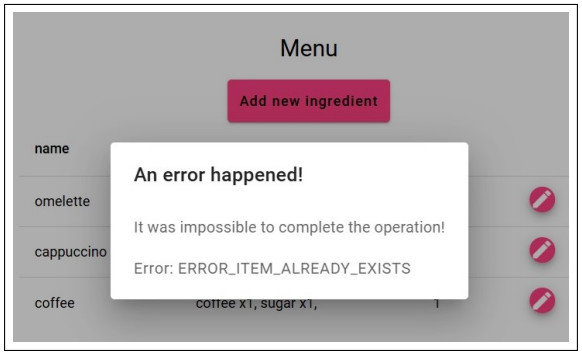

Figure 13: Error dialog

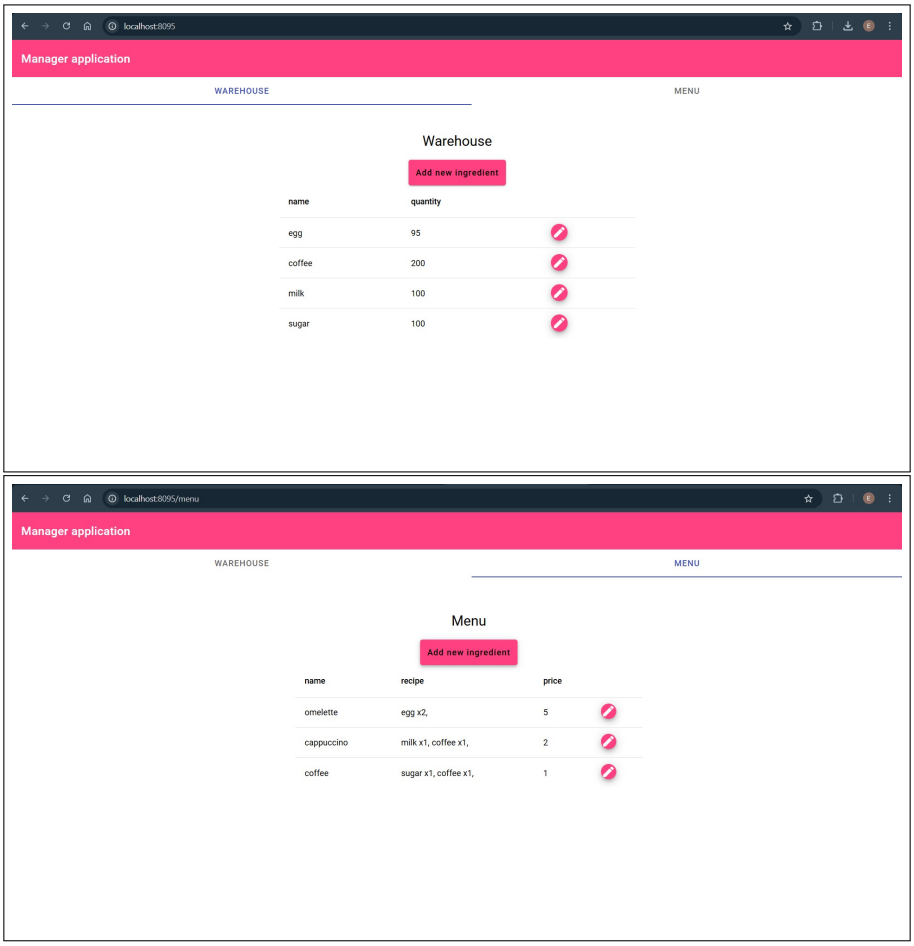

Figure 14: Warehouse and menu pages

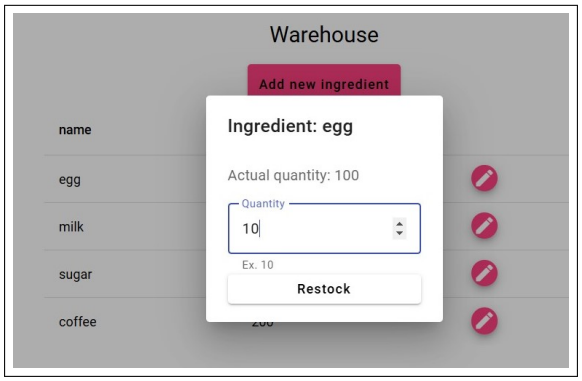

Figure 15: Form used to restock an ingredient quantity

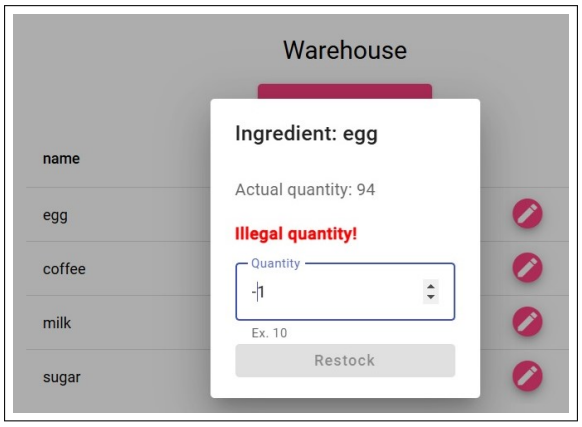

Figure 16: Error message when an illegal quantity is selected

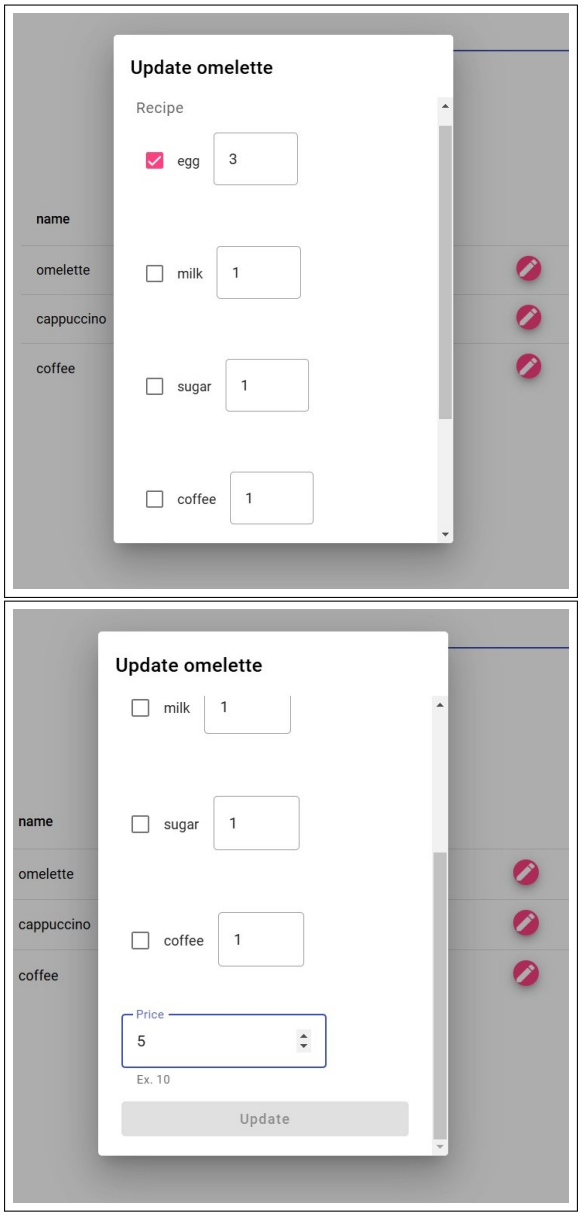

Figure 17: Form used to update an item

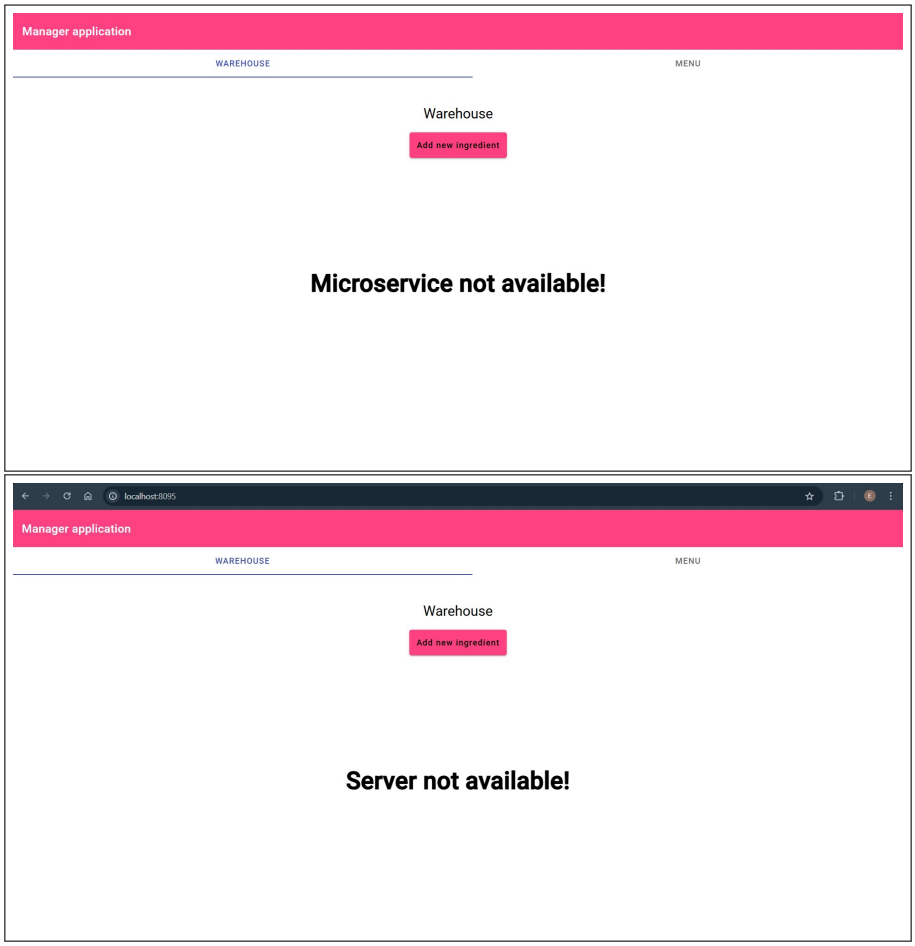

Figure 18: Microservice/server not available

7.2 customer-application

At the beginning, if the warehouse/menu databases are empty or there are no available items, an error message will be displayed to the user (figure 19).

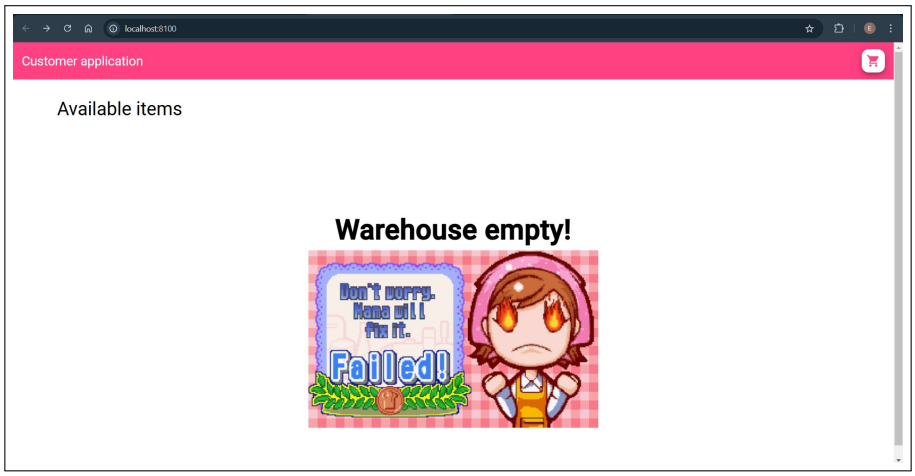

Figure 19: customer-application home page when the warehouse database is empty

The home page shows all the available item a customer can order, with their recipe. The user can select the quantity of items he wants to order, and then can add them to the cart with the "Add to cart" button (figure 20).

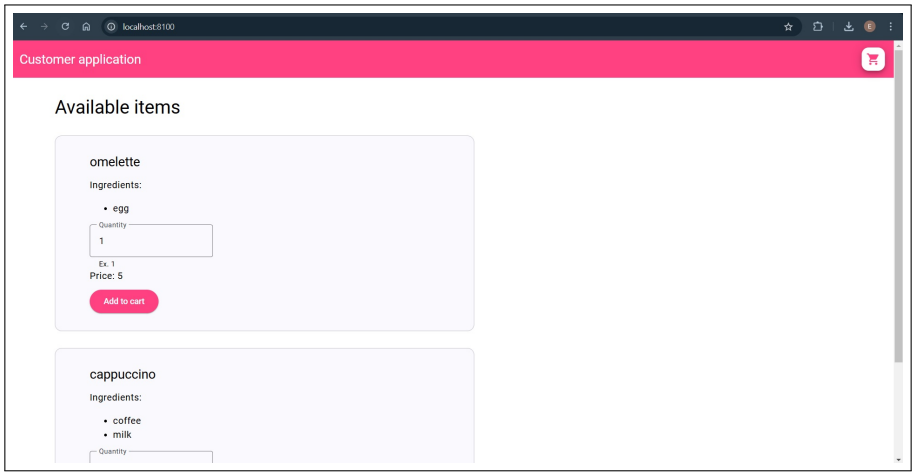

Figure 20: customer-application home page

Clicking the "cart" button in the navbar, it is possible to see the order recap (figure 21). In this page it is possible to modify the order removing items and changing their quantities.

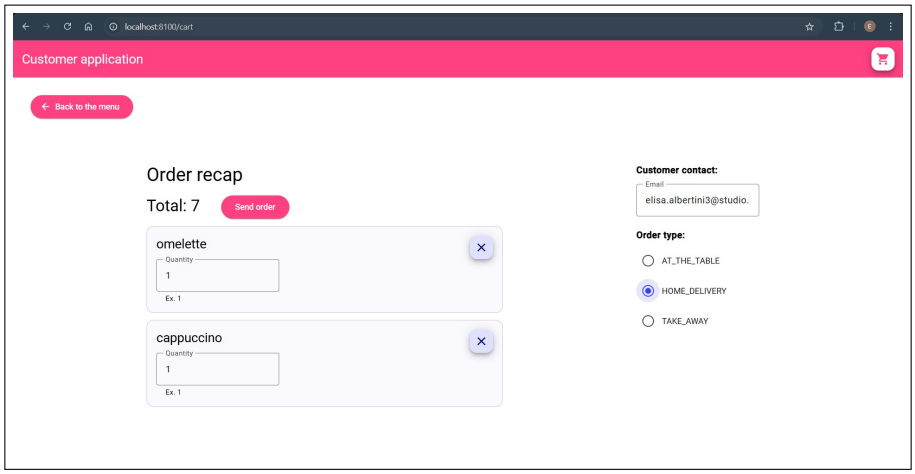

Figure 21: Order recap page

Recap page also displays some fields that have to be mandatory filled to send an order, such as the customer email and the type of orders.
Once an order is sent, if it's possible to complete it, a notification will be shown to the user (figure 22), otherwise it will be displayed an error message (figure 23).

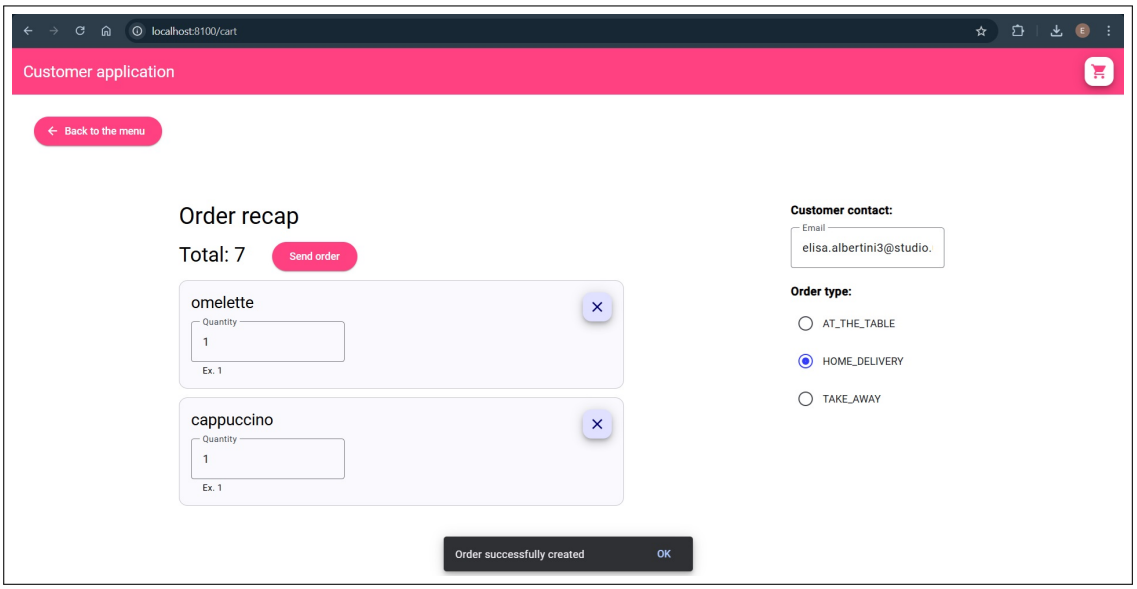

Figure 22: Notification of an order successfully created

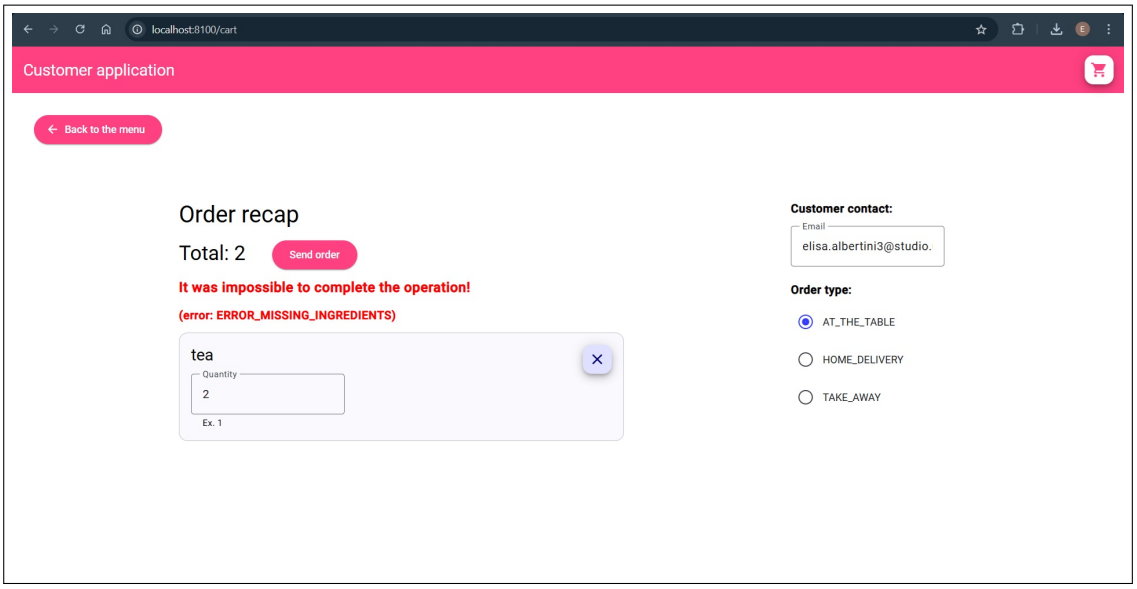

Figure 23: Error message if it's not possible to create an order

7.3 EmployeeApplication

At the beginning, if the orders database is empty, an error message will be displayed to the user (figure 24).

The application shows the orders in real time (figure 25).

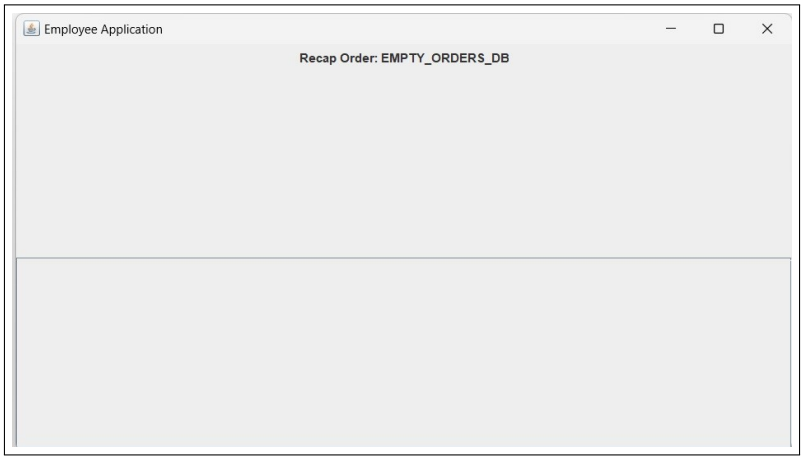

Figure 24: EmployeeApplication when the orders database is empty

It is possible to modify the state of the orders selecting the new one and clicking on the "Choose state" button (figure 26). If the new state selected is illegal a message will be shown (figure 27). When an order became ready, an email will be sent to the user (figure 28).

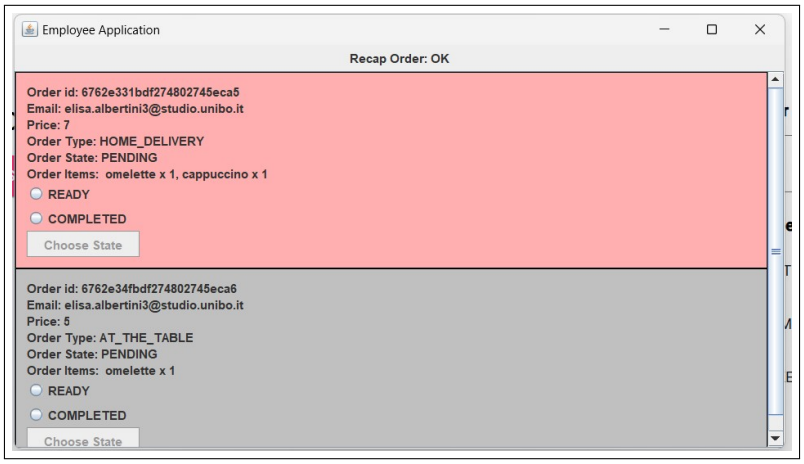

Figure 25: EmployeeApplication

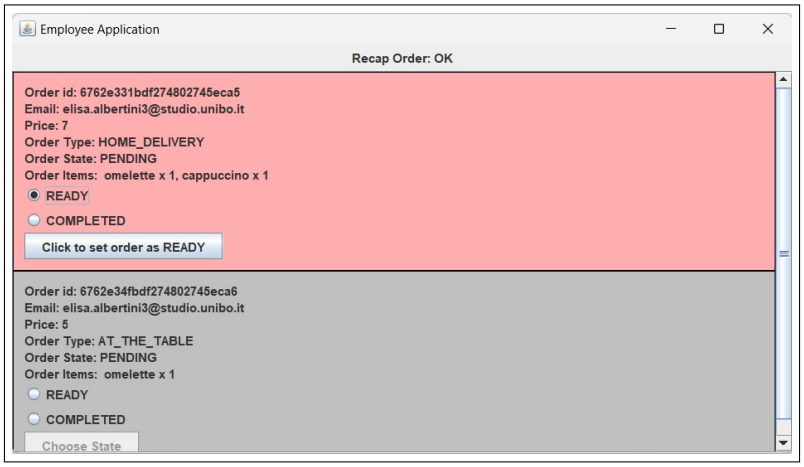

Figure 26: EmployeeApplication when the user changes the state of an order

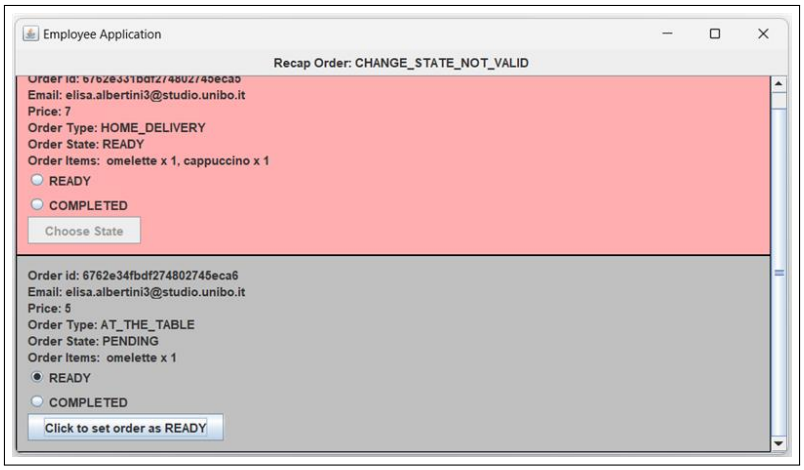

Figure 27: EmployeeApplication when the new state is illegal

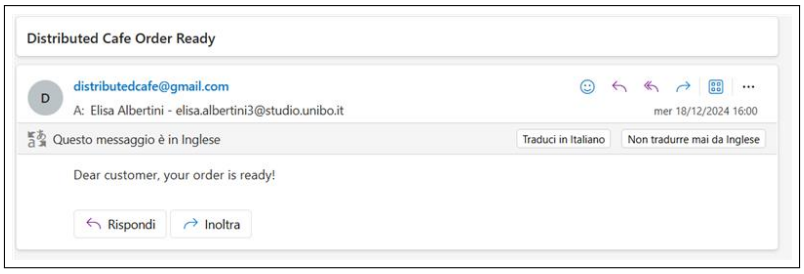

Figure 28: Email sent to the user

8 Conclusions

In conclusion, we developed a distributed system divided in back-end and front-end in communication between each other, both composed by different components, exploiting asynchronous programming.

We challenged ourselves by composing the back-end of three independent microservices, each of them with their own database, exposing REST API and of a server that communicates with the front-end exploiting web socket and that communicates with microservices by calling their API. We also designed each microservice in order to manage a different portion of the domain to have more modularity. We dealt also with real time communications and with notifications thanks to web socket.
We exploit the server to check if the microservices are running. In case the server itself goes down the front-ends will notify it to the user.

Our choice for the front-end was to realize two web applications, implemented with Angular, and a Java application, meant to be used by customers, managers and employees.

We also learned how to implement containerization using a container for MongoDB and each component of our system, except for EmployeeApplication. Lastly, we exploit Docker Compose to easily deploy the containers.

References

[1] OpenJS Foundation. Express: Fast, unopinionated, minimalist web framework for node.js. https://expressjs.com/. Accessed on 2024.

[2] Matt Zabriskie and contributors. Axios: Promise based http client for the browser and node.js. https://axios-http.com/, 2014. Accessed on 2024.

[3] Eclipse Foundation. Eclipse vert.x™ reactive applications on the jvm. https://vertx.io/community/. Accessed on 2024.

[4] Inc. MongoDB. Mongodb: The developer data platform. https://www.mongodb.com/it-it. Accessed on 2024.

[5] Google. Angular is a web framework that empowers developers to build fast, reliable applications. https://angular.dev/overview, 2010. Accessed on 2024.

[6] Samchon & Contributors. Typia, superfast runtime validator. https://typia.io/. Accessed on 2024.

[7] Inc. Docker. Docker, accelerated container application development. https://www.docker.com/. Accessed on 2024.