FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distribution and Integration Technologies
4th year of the Integrated Masters in Informatics and Computing Engineering (MIEIC)

An enterprise distributed system
Book store and warehouse system

Group elements:

Diogo Luis Rey Torres - 201506428 - up201506428@fe.up.pt
Rui Emanuel Cabral de Almeida Quaresma - 201503005 - up201503005@fe.up.pt

May 26, 2019

Index

1.Introduction

The developed project consists of an enterprise distributed system capable of managing a bookstore in which it's responsible for managing sells, orders and stock. It was developed using the principles of a service-oriented architecture (SOA) and consists essentially of two large modules: the store module and the warehouse module.

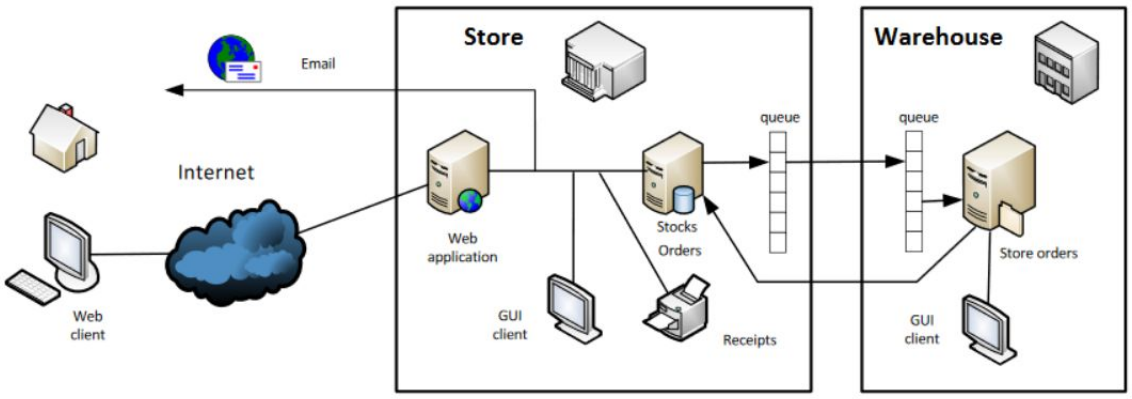
A bookstore needs to coordinate the sells (made in the physical store) and the orders (coming from the online store or the physical store), allowing them to be quickly processed to the client. When a user makes a sell/order, the bookstore needs to process the request. Processing a request consists of checking if the stock is sufficient to satisfy the sell/order, if not a request stock to the warehouse is made, and after that it's delivered to the client. With this in mind, the system implements all the necessary requirements for its operation presenting them in an intuitive and pleasant way for the final user (clients and employees).

In the next sections, the application will be described in finer detail, namely its architecture, implementation details and relevant issues. This report concludes with a small reflection about the work done and possible future enhancements.

2. Architecture

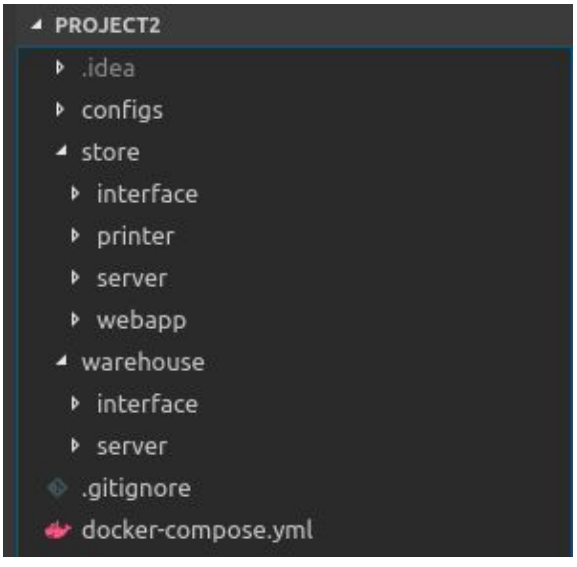
2.1 Description

This application is based on a Service Oriented Architecture where services are provided to the other components by application components, through a communication protocol over a network. That allows the creation of a system from a collection of isolated services, each of which owns their data, and is independent, scalable and resilient to failure. In addition to this architecture, there is also the client-server architecture (more specifically the 3-Tier Architecture) in the communication between the terminals/webApp (clients) and the store/warehouse (servers).



Subtitle 1: System Architecture

The used architecture consists of two modules: the store module and the warehouse module. The store module is composed by a web application, a GUI client, a printer and the store server with its own database. The warehouse module is only composed by a GUI client and the server with its own database.



Subtitle 2: Solution Overview

Each of the services (except the GUIs) is containerized that allows the application to run quickly and reliably from one computing environment to another, as well as allowing each container to operate independently from the others preventing interdependencies and also safeguarding from a single point of failure.

The communication between these two modules is done through message queues, where the store sends messages to the warehouse queue and when the warehouse responds it also sends a message to the store queue. This allows an asynchronous communication and even if one of the servers is offline when it is reconnected it will consume the messages that are persisted in the queue and process those requests accordingly, creating a more fault tolerant system.

The communication to the servers (api's) is made through a reverse proxy server that serves as intermediary to forward the requests for content from multiple clients (web client and GUIs) to the corresponding servers. This provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers.

Subtitle 3: nginx.conf file

It also allows the servers to follow the REST architecture that provides interoperability between computer systems on the Internet. Apart from this type of communication, they also provide a communication over websockets that allows real-time information to be passed to the clients who subscribed to certain channels waiting for events.

2.2 Modules and Services

2.2.1 Store Module

2.2.1.1 Web Application

This application is based on *React* framework that presents to the user the online bookstore. It communicates with the store server through REST calls.

This communication is assured via "*axios*", a promised based HTTP client for nodeJS.

The Web App is divided in components either pages or parts used to be listed.

There are also helper classes which have methods to deal with Auth, Books, Clients and Orders.

The main page, App.js, is wrapped with a *AuthWrapper* which purpose is to check if the user is allowed to navigate to that page (if he/she has logged in). If not the user is sent to the login page. On the main page the user has a list of his/her orders, a button to create a new order and on the app bar, a button to logout.

The login page consists of a login form and a link to the register page.

The register page consists of a registration form and a link to the login page.

The createOrderPage is where the user may create a new order. In order to do so, the user must be associated to a client. For this purpose two fields (name and address) are in the order creation form so that he/she becomes associated to a client. There is a list of the books available and a field to select the quantity desired. When all fields are filled the user may create a new order.

The helper files have the methods to perform actions on the server using axios.

2.2.1.2 GUI Client

This application is based on *C#* and *WinForms* that presents to the employee the bookstore. It communicates with the store server through REST calls (*RestSharper*) and updates the information in real-time through websockets (*Pusher*).

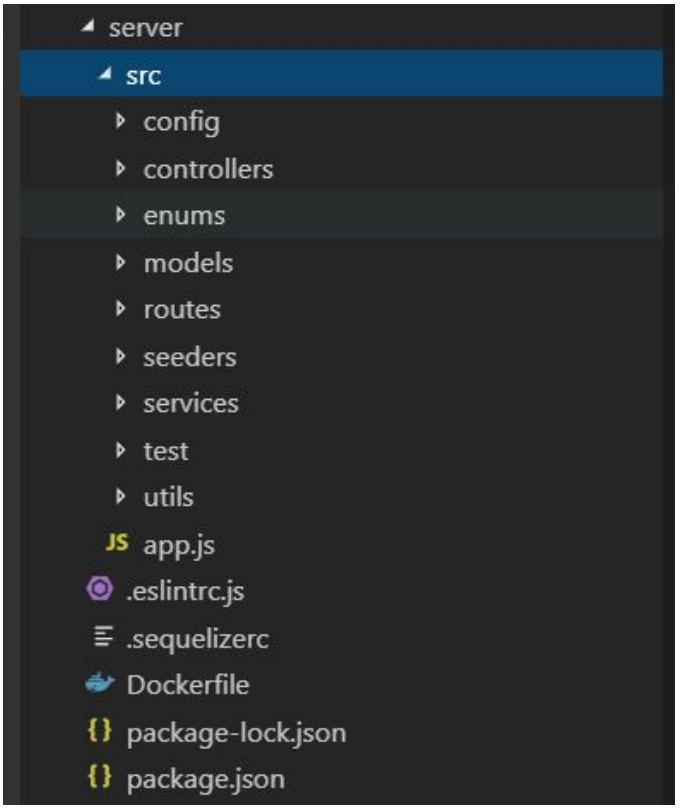
There are multiple windows including a *LoginWindow*, a *StoreWindow*, a *NotificationsWindow*, a *StatisticsWindow*, a *ClientWindow* and windows to show the sells and orders made in the store.

2.2.1.3 Printer

This application consists only of HTML, CSS and Javascript that provides a simple way to simulate the printer. The printer receives the receipts through the websockets (*Pusher*) from a specific channel (*store*) binding a specific type of notifications (printer_invoice).

2.2.1.4 Server

The server is based on *ExpressJs* framework with the following structure:



Subtitle 4: Server's folders

This structure allows a good division between models, logic, services and routes that follows good programming practices. The server has a database in *postgres* that saves all the information necessary to manage the store. This database contains the following models: Book, Client, Order, ReceiveStock, Sell and User.

The interaction between store and warehouse is made when the store publishes messages to the warehouse queue called: '*request_stock'* and consumes messages from the warehouse through the '*receive_stock'* queue. Also, all the changes in database (specifically creates and updates on the models) are transmitted to the store GUI through websockets in the '*store'* channel. Also, each create/update model has a specific type to allow better recognition of events when they are subscribed by a specific client (in this case the store GUI).

```
const messageType = {
  createClient: 'create_client',
  updateClient: 'update_client',
  updateBook: 'update_book',
  createOrder: 'create_order',
  updateOrder: 'update_order',
  createSell: 'create_sell',
  updateSell: 'update_sell',
  createReceiveStock: 'create_receiveStock',
  updateReceiveStock: 'update_receiveStock',
  printInvoice: 'print_invoice',
};
```

Subtitle 5: Message types used in Pusher notifications

The store server when creating or updating an order automatically sends an email to the client. This is done through a plugin called *nodemailer* that sends messages using a SMTP protocol. In this project there are two Nodemailer transporters configured : one through *mailtrap.io* and the other through *Gmail*. The first is the default that allows catching emails sent by the server for testing and debug purposes by not sending the emails directly to clients. The second is for when the application is production-ready, to actually send the emails to customers. Also, the implementation of the email service on the store server has a queue to push the emails. This queue is necessary for optimization purposes and, since the SMTP clients only allow 2 emails to be sent every 10 seconds, there's an internal counter that automatically triggers a timeout whenever necessary so as to not go over that limit.

Finally, the store server has a seed database for test purposes that allows during the development to easily test the application and new features.

2.2.2 Warehouse Module

2.2.2.1 GUI Client

This application is similar to the one on the Store module. Its windows include a *LoginWindow* and a *WarehouseWindow*.

2.2.1.2 Server

The warehouse server follows the same structure as the store server (2.2.1.4). However, the database has only the following models: Book, Request and User.

The interaction between warehouse and store is pretty much the same as described in the 2.2.1.4 section but the warehouse publishes messages to the store queue called: '*receive_stock'* and consumes messages from the store through the '*request_stock'* queue. Also, all the changes in database are transmitted to the warehouse GUI over websockets through the *'warehouse'* channel.

```
1  const messageType = {
2    updateBook: 'update_book',
3    createRequest: 'create_request',
4    updateRequest: 'update_request',
5  };
6
```

Subtitle 6: Message types used in Pusher notifications

Finally, the warehouse server has a seed database too.

2.2.3 Common code on store and warehouse GUIs

The abstract controller was created to avoid code repetition on clients, thus containing the most common methods and variables used by them and inherited when the client controllers are created.

The *Pusher* is being used to receive the requests via webSockets and update the GUIs on real time.

Different pushers are created for each messageType. These subscribe either the store channel or the warehouse channel and are binded to each one upon corresponding window opening and unbinded on window closing.

The action to be executed when the pusher receives a message is passed to it as a Delegate.

3. Technologies Used

In this chapter, it will be summarized the functionalities of each technology used.

3.1. Docker + Docker-compose

Docker is ideal to develop a system with container-based apps that brings consistent development environments for the entire team for developers, making it easier to deploy and isolate the applications.

Compose is a tool for defining and running multi-container Docker applications. With Compose, we use a configuration file that allows us to configure and start all the services with a simple command.

3.2. Nginx

Nginx is a web server which can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.

3.3. RabbitMQ

RabbitMQ is the most widely deployed open source message broke that implements the Advanced Message Queuing Protocol (AMQP).

3.4. Pusher

Pusher is a hosted service that makes it *super-easy* to add real-time data and functionality to web and mobile applications.

Pusher maintains persistent connections to the clients - over WebSocket if possible and falling back to HTTP-based connectivity - so that as soon as the servers have new data that they want to push to the clients they can do, instantly via Pusher.

3.5. Mailtrap.io

Mailtrap simulates the work of a real SMTP server. It isolates the staging emailing from production and eliminates any possibility of a test email to land in a real customer's mailbox.

3.6. Reactjs

Reactjs is a JavaScript library for building user interfaces that makes it painless to create interactive UIs and allows the creation of encapsulated components that manage their own state, then compose them to make complex UIs.

3.7. C#

C# is a general-purpose, modern and object-oriented programming language pronounced as “C Sharp”. It was developed by Microsoft.

With this language are used some libraries, the most relevant are:

- Pusher-websocket-dotnet
    - This is a .NET library for interacting with the Pusher WebSocket API.
- RestSharp
    - Simple REST and HTTP API Client for .NET
- Newtonsoft.Json
    - Popular high-performance JSON framework for .NET

3.8. Nodejs (Expressjs Framework)

Express.js is a web application framework for Node.js. It is designed for building web applications and APIs.

With this framework are used some libraries, the most relevant are:

- Passport
    - Passport's sole purpose is to authenticate requests. A JWT Strategy is used to authenticate users through the API. We also save the password with bcrypt in database.
- Nodemailer
    - Allows sending emails from Node.js.
- Pg + Sequelize
    - Sequelize is a promise-based Node.js ORM for SQL databases and allows a better database manipulation.
- Pusher
    - Allows to trigger notifications through websockets.

3.9. Postgres

PostgreSQL is a relational database management system (RDBMS) emphasizing extensibility and technical standards compliance. It is designed to handle workloads ranging from one machine to large data warehouses or Web services with many concurrent users.

4.Functionalities included

4.1. Store

- Login as an employee
- View a list of the clients
- View a list of the books in the store
- View a list of all orders
- View a list of all sells
- View a client's information as well as his/her orders and sells
- View the statistics including the top books sold and the total sold
- Add a new client
- Sell a certain amount of a book to a client
- Order a certain amount of a book for a client
- View a list with the notifications of stock received and stock to deliver and their state
- Deliver an order once it is on the Dispatch state

4.2. Warehouse

- Login as an employee
- View a list of the books in the warehouse
- View a list of books requests and their state
- Ship a request that is on the Waiting state

4.3. WebApp

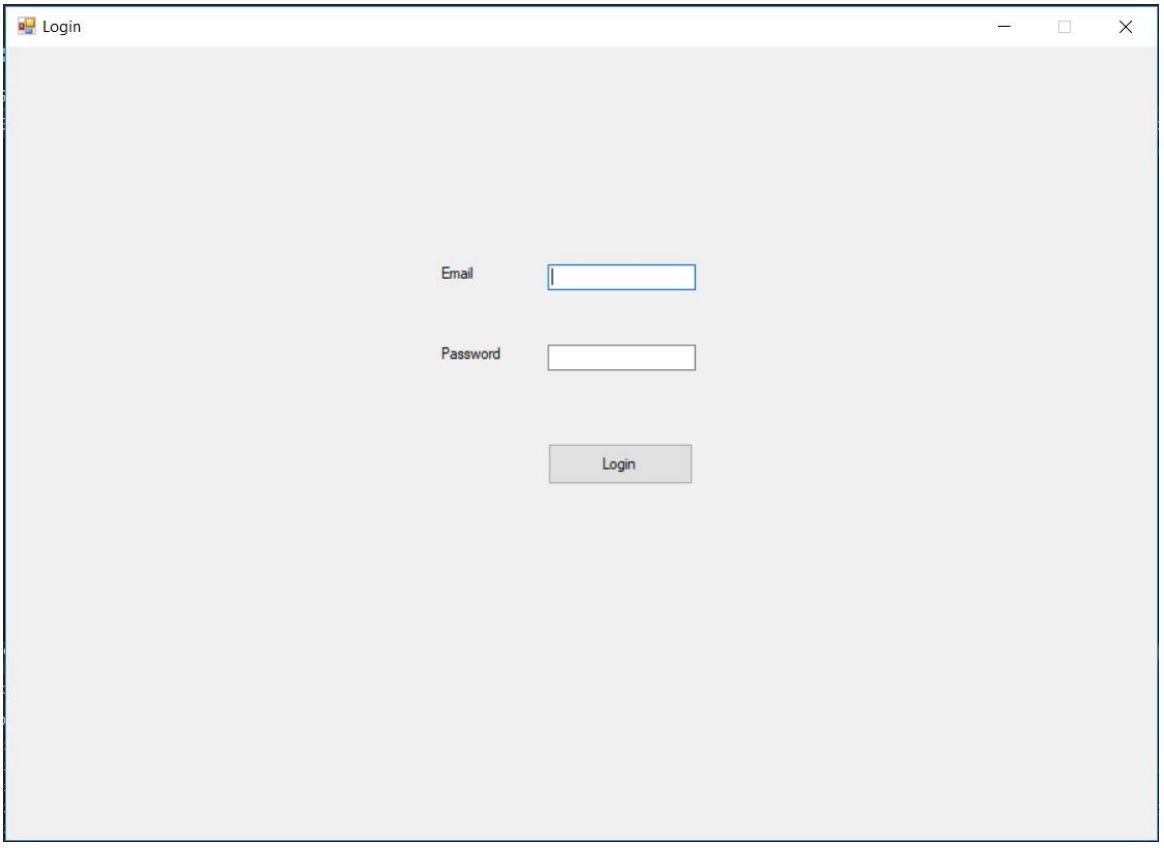
- Login as a user
- Register as a new user
- View a list of orders
- Create a new order
- View a list of books
- Associate the user to a client

All requested features have been implemented as well as some extra features.

5.Screen Captures illustrating the main sequences of use

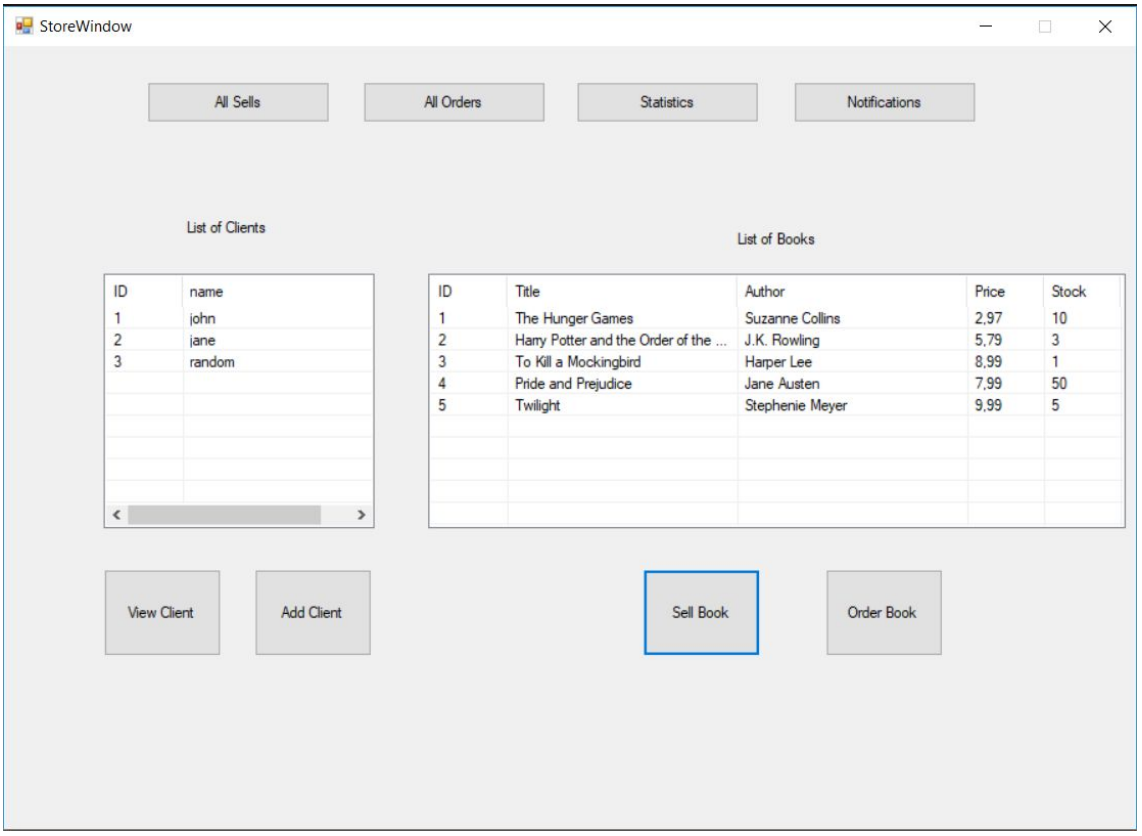
5.1. Store

The App flow starts by opening the *LoginWindow*.



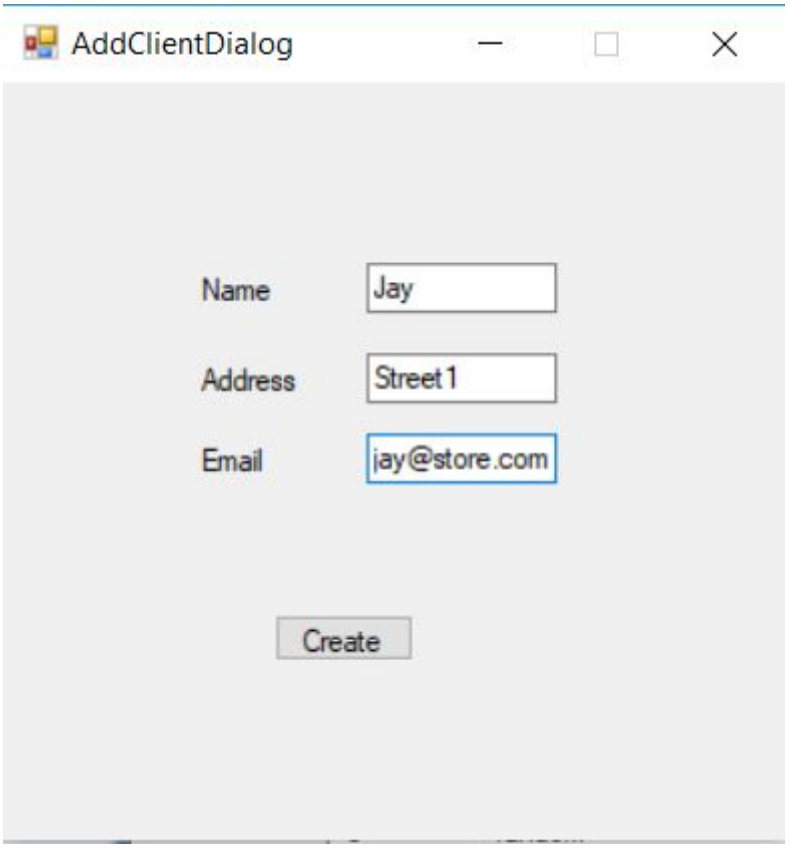
Subtitle 7: LoginWindow

After logging in, the employee is presented with the *StoreWindow*.



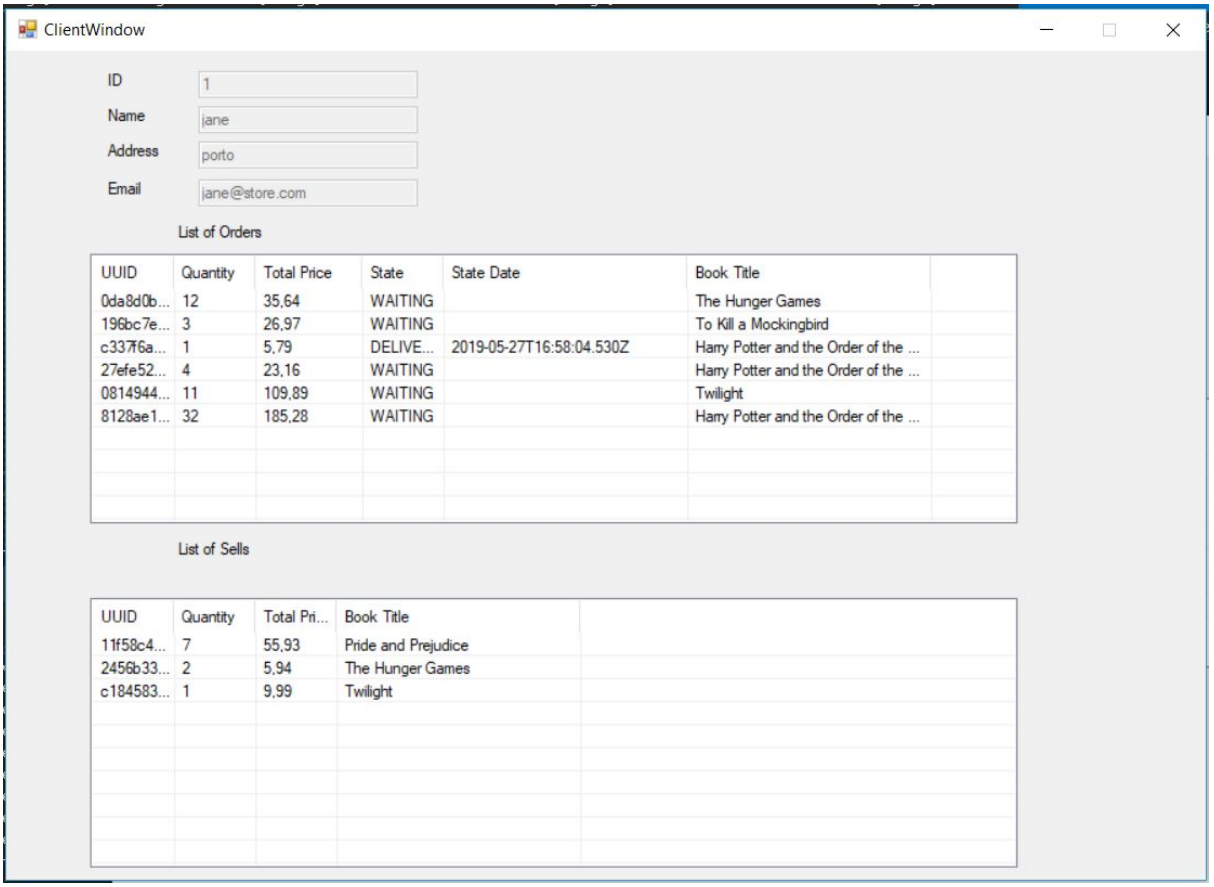
Subtitle 8:StoreWindow

Clicking on Add Client, the client creation dialog opens.



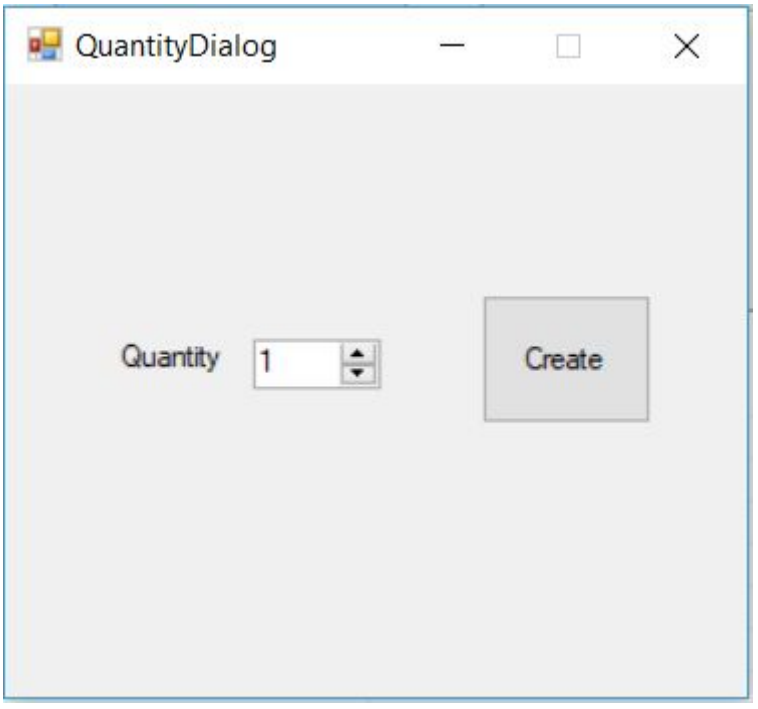
Subtitle 9:AddClientDialog

Picking one client and clicking on View Client, the *ClientWindow* is opened.



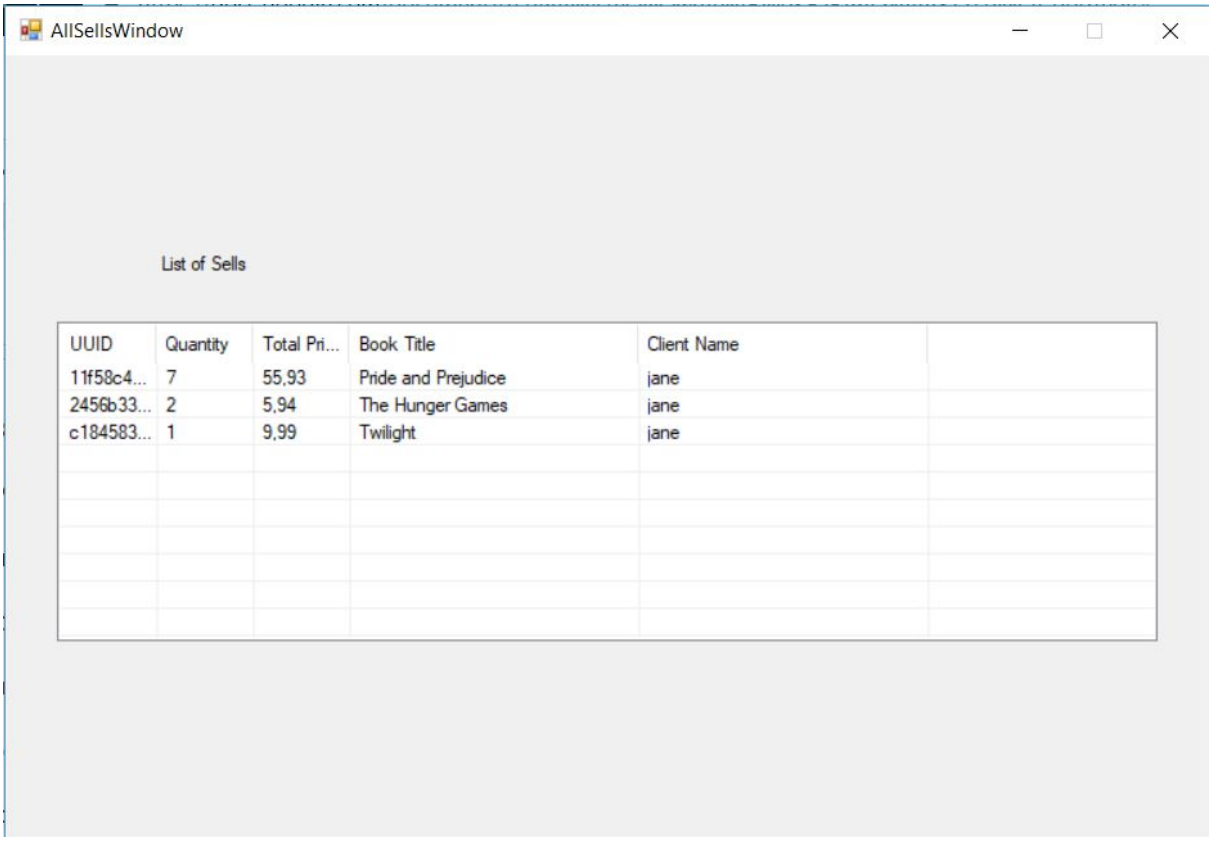
Subtitle 10:ClientWindow

Picking one client and a book, and clicking on Sell Book or Order Book, the *QuantityDialog* opens. After choosing the quantity, an order or a sell (accordingly) will be created.



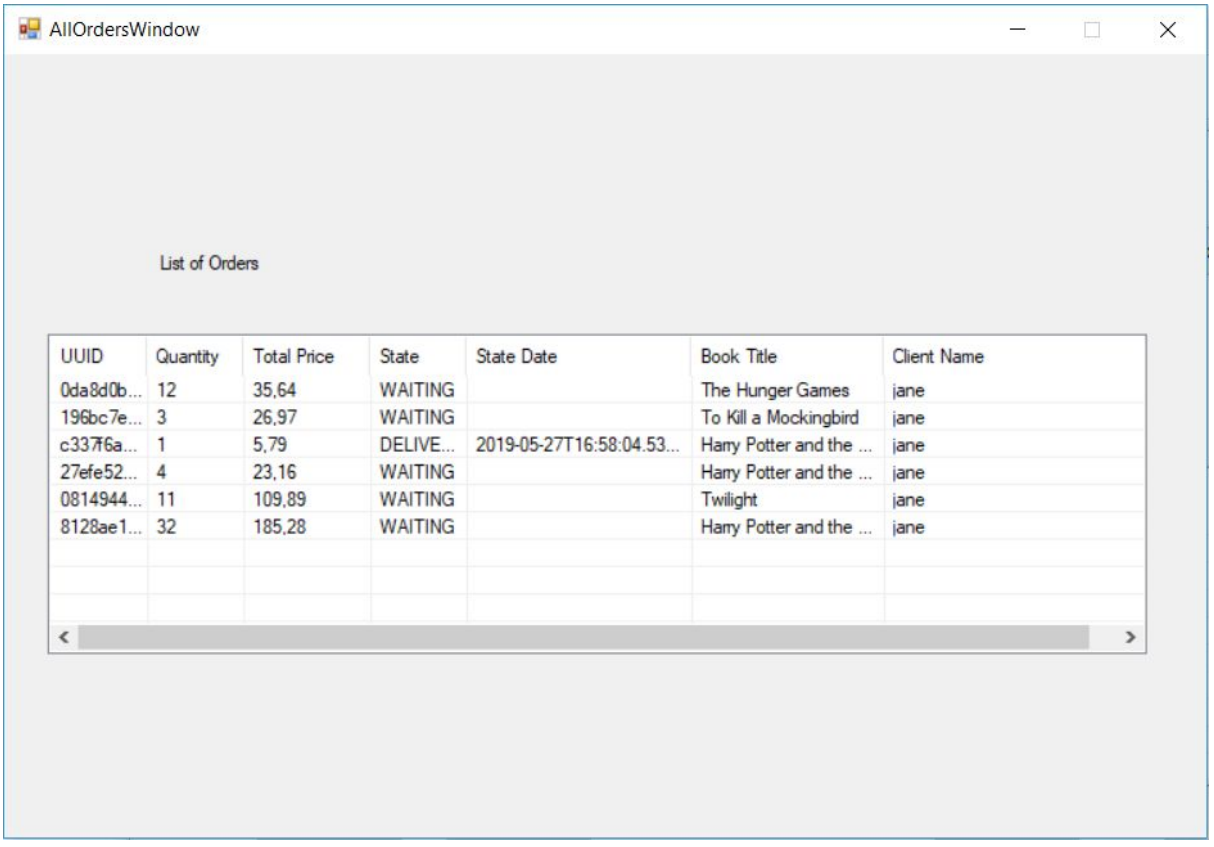
Subtitle 11:QuantityDialog

Clicking on All Sells, the *AllSellsWindow* opens.



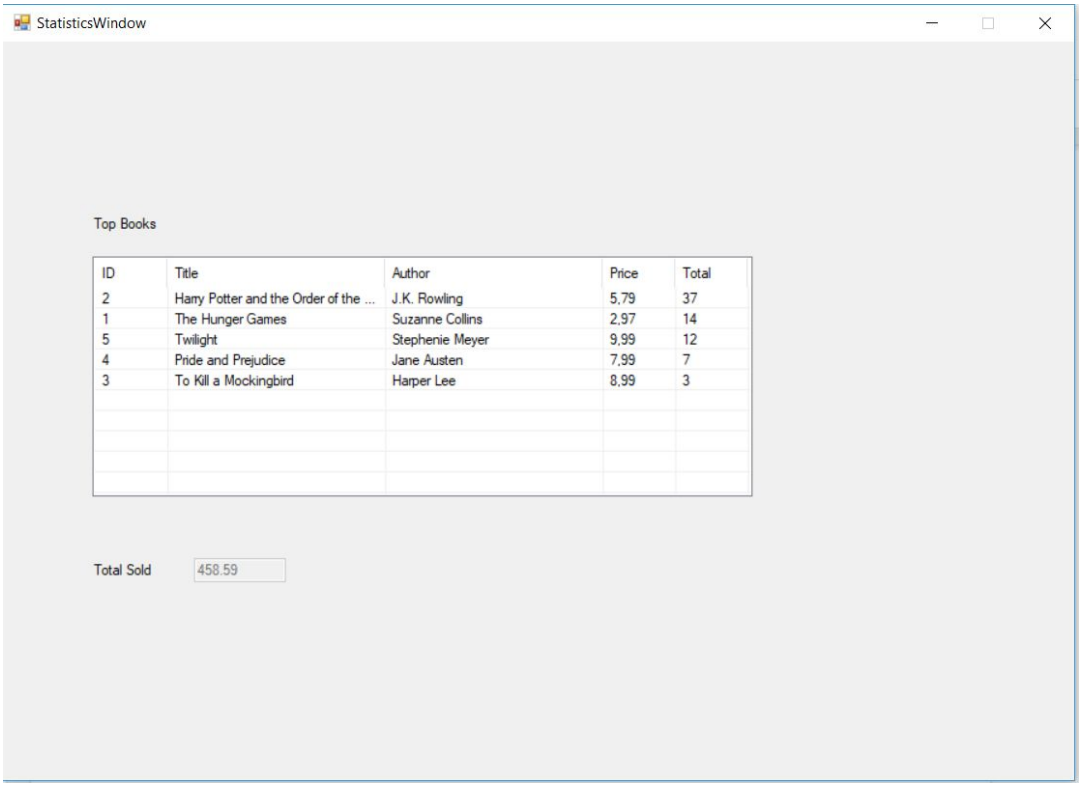
Subtitle 12:AllSellsWindow

Clicking on All Orders, the *AllOrdersWindow* opens.



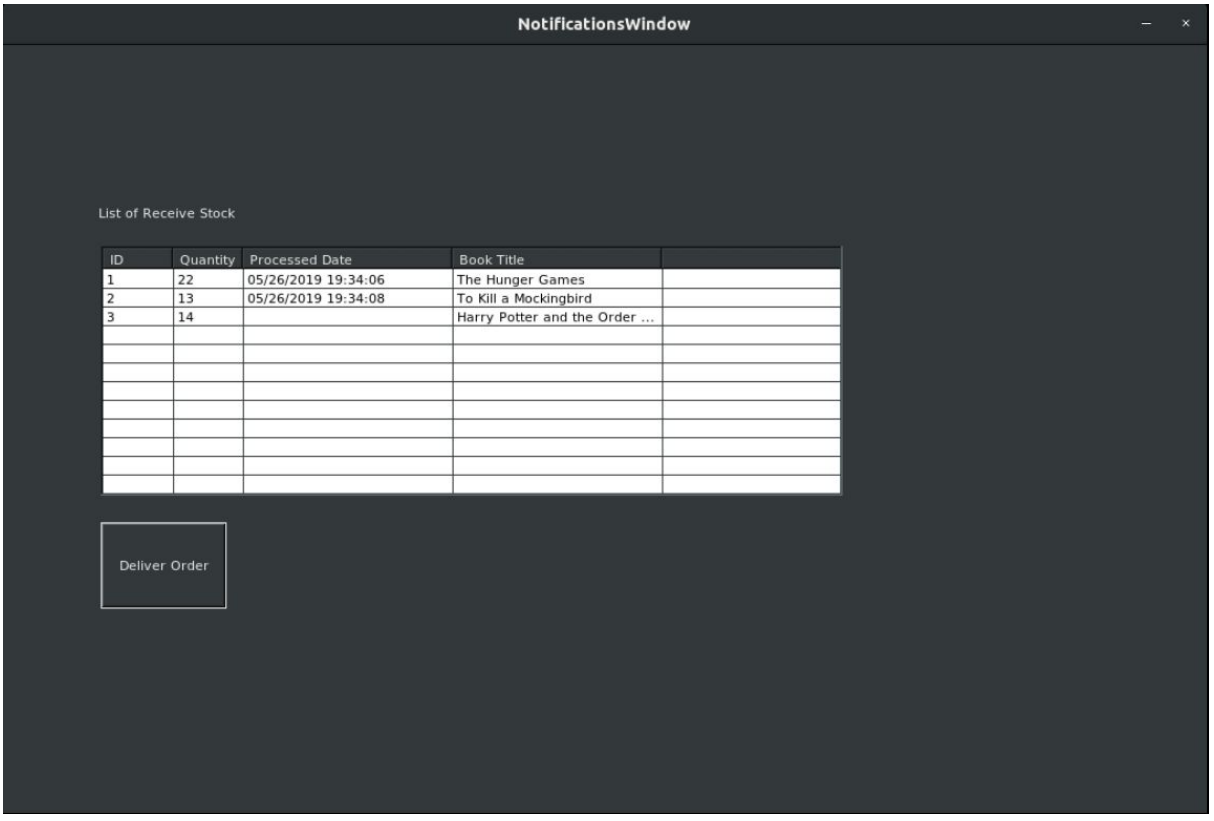
Subtitle 13:AllOrdersWindow

Clicking on Statistics, the *StatisticsWindow* opens.



Subtitle 14:StatisticsWindow

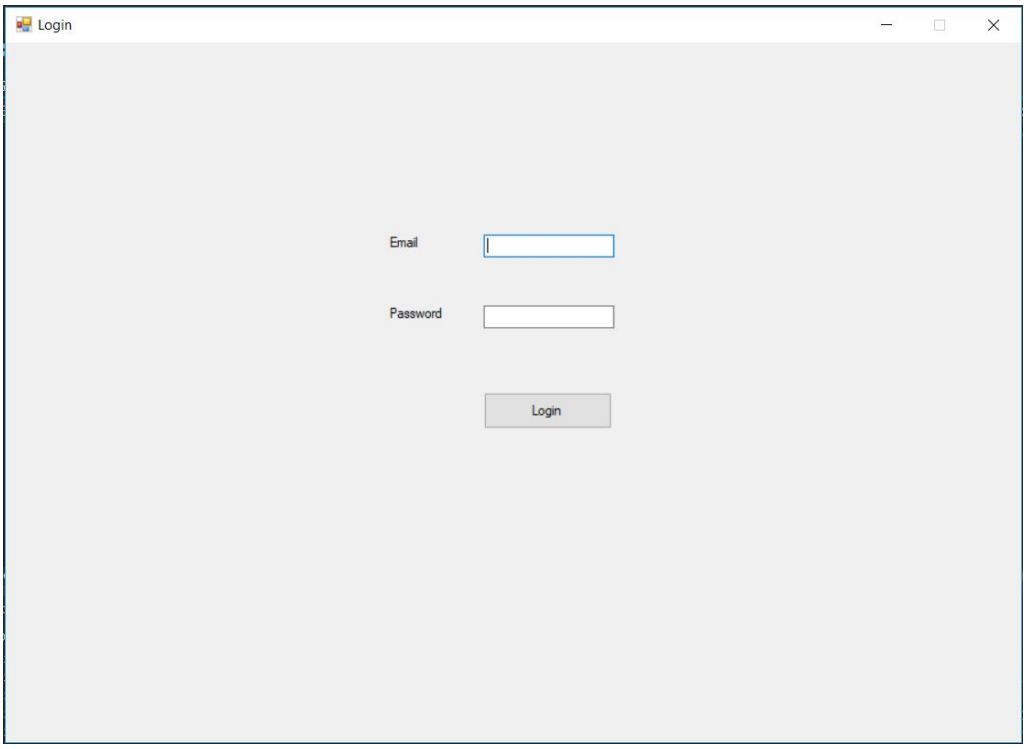
Clicking on Notifications, the *NotificationsWindow* opens. Picking one request of stock and clicking on Deliver changes that request to the state Delivered.



Subtitle 15:NotificationsWindow

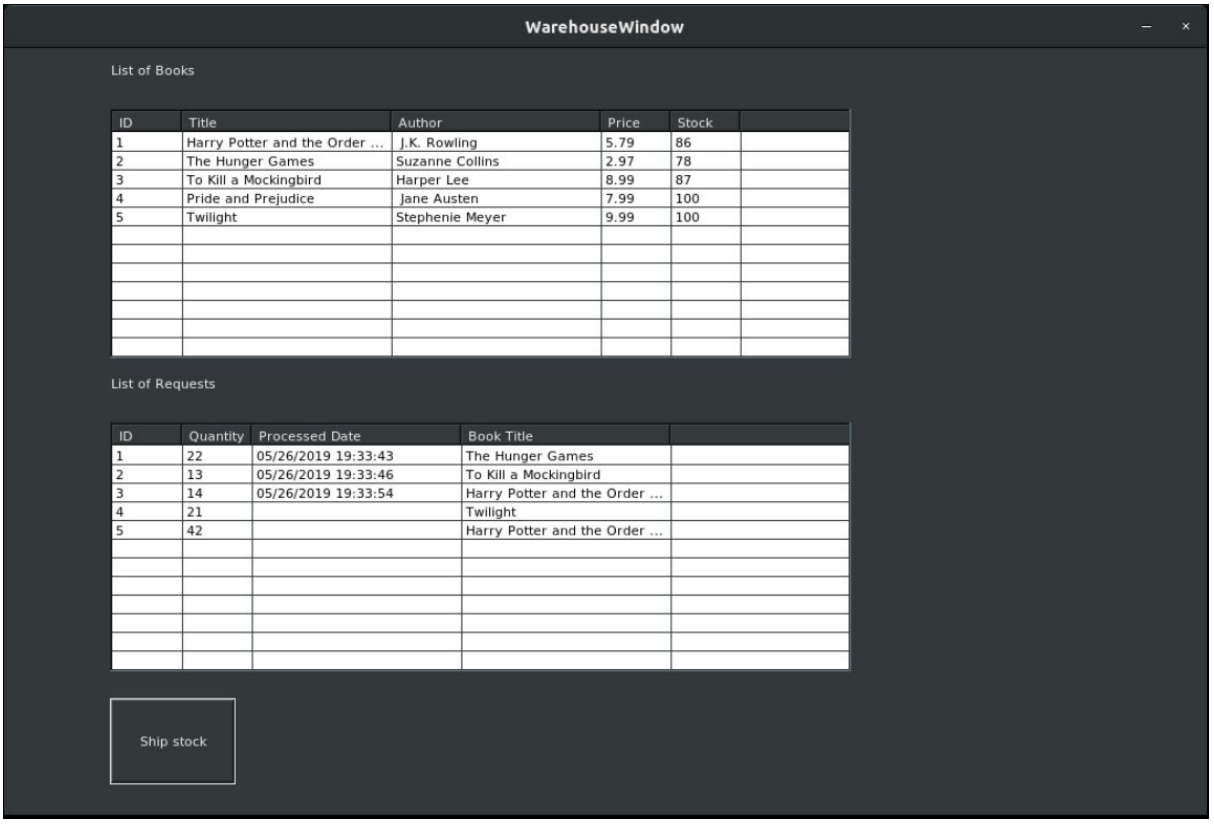
5.2. Warehouse

The App flow starts by opening the *LoginWindow*.



Subtitle 16: LoginWindow

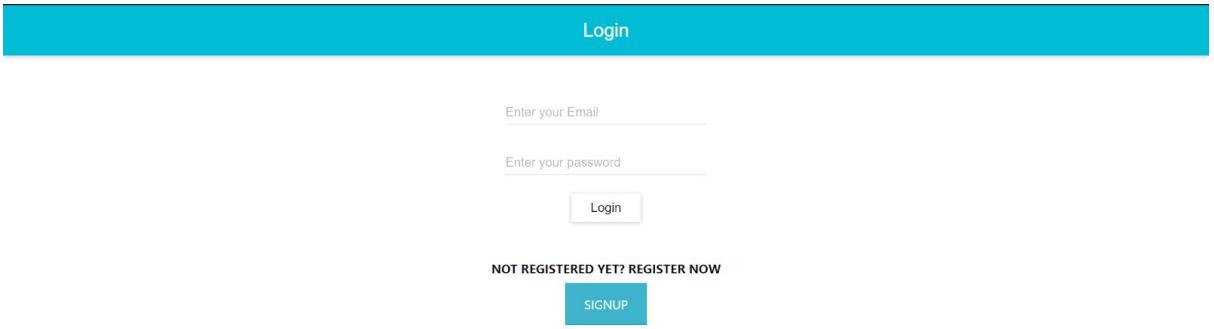
After logging in, the employee is presented with the *WarehouseWindow*. After picking one request it may be shipped through pressing Ship Stock.



Subtitle 17: WarehouseWindow

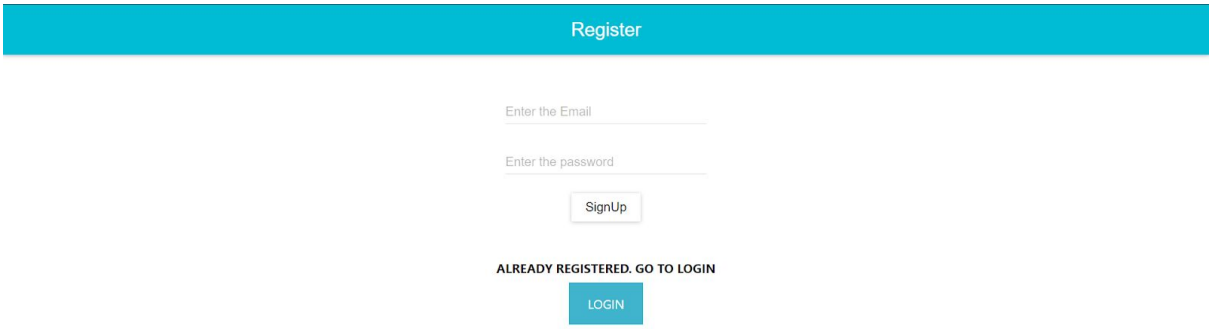
5.3. WebApp

The first screen shown is the Login.



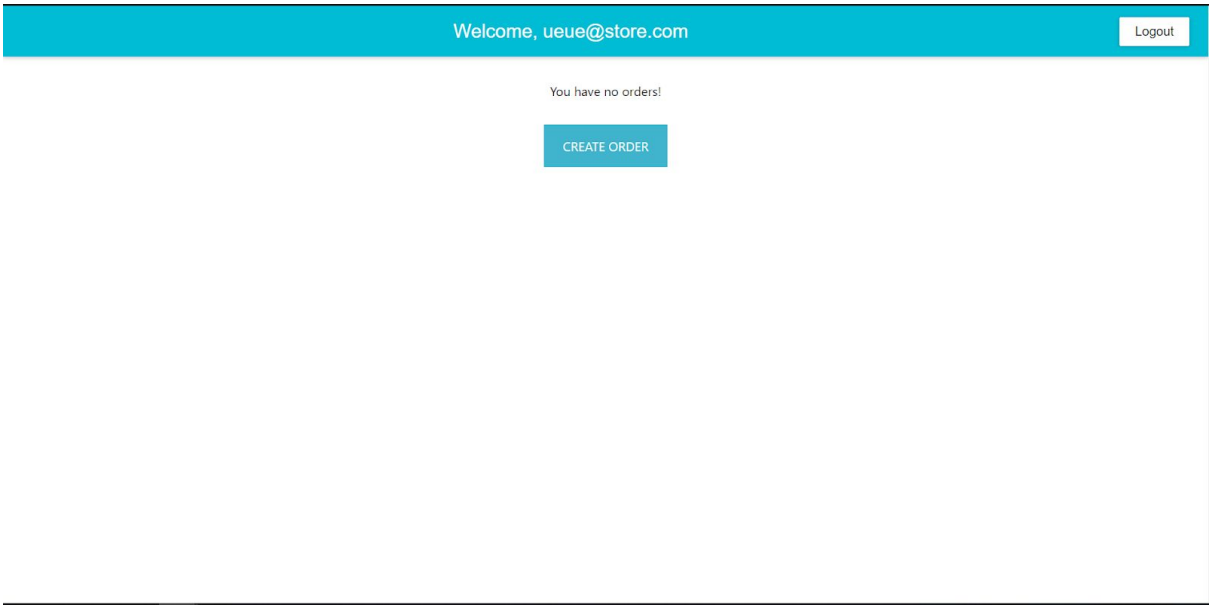
Subtitle 18: Login

From there we can also access the Register screen.



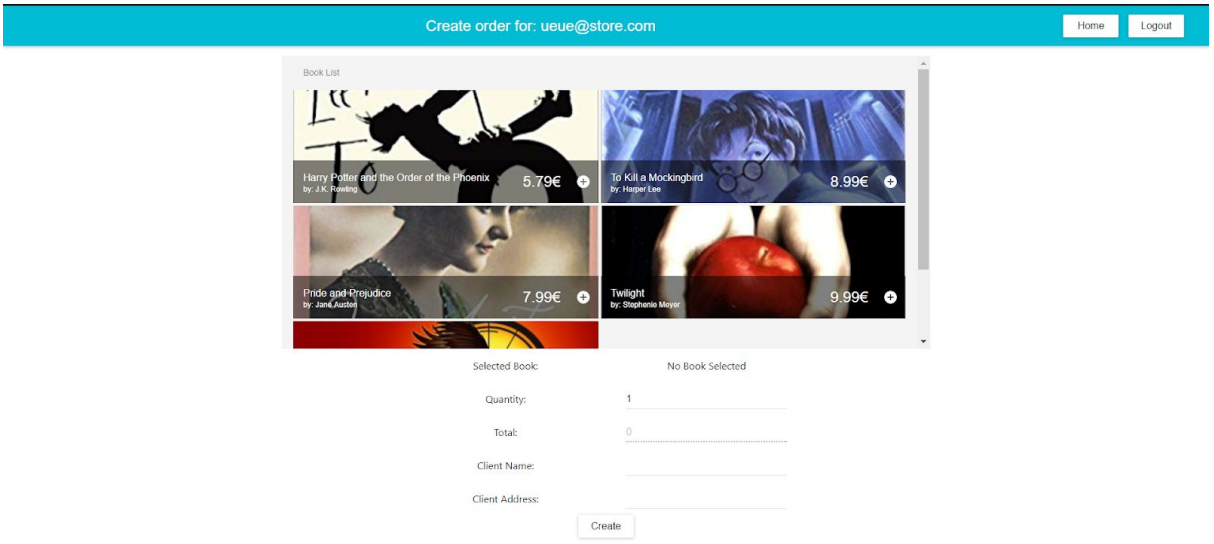
Subtitle 19: Register

When the user enters, the main page is shown.



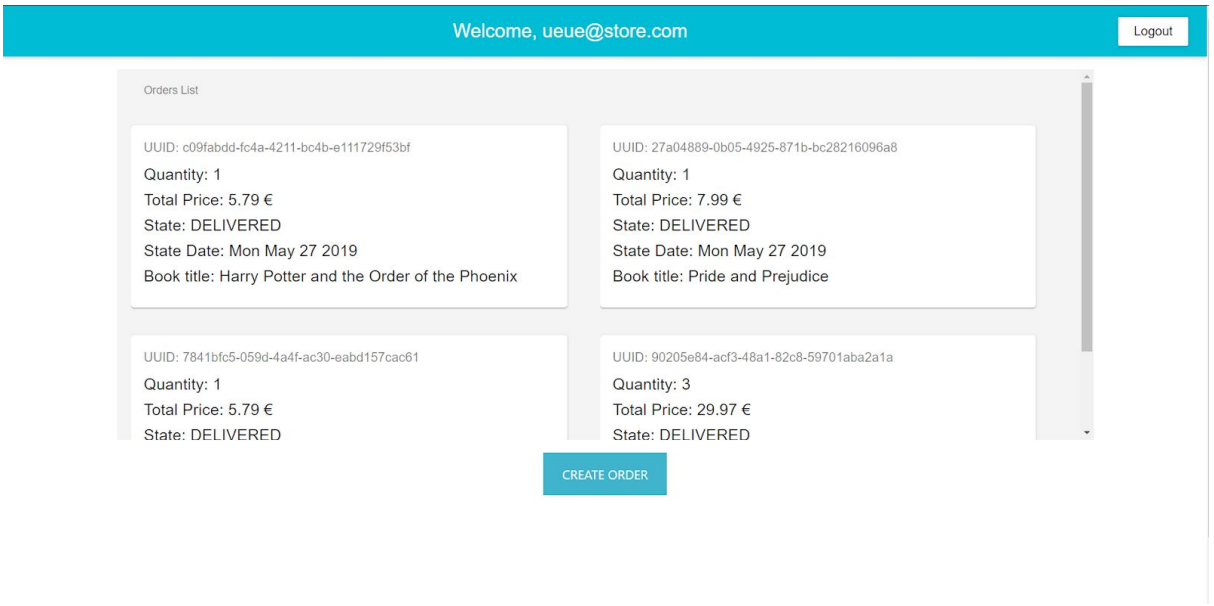
Subtitle 20: Main Page

Clicking on Create Order, the user goes to the *CreateOrderPage*. In here he/she may select a book and the desired quantity. If it is the first order, then his/her name and address are also requested.



Subtitle 21: CreateOrderPage

After that, the main page now has the list of orders.



Subtitle 22: Main Page

6.Conclusion

With this work, the group realized the importance of a well structured architecture in the development process in order to achieve a good final result. We are able to learn the basic operation of applications developed with message queues and following a service oriented architecture as well as the main problems associated with them (compatibility between systems).

The final project provides a set of applications/services that can interact with each other in order to create an enterprise distributed system capable of managing a bookstore in which it's responsible for managing sells, orders and stock. The stock is provided by a warehouse that's also present in the system.

Despite this, as always, there can be enhancements done so as to further elevate its quality and performance. In this case, we can develop automated tests to verify that all functionalities are operational, as well as optimizing the requests through the message queues, creating a cron job to update automatically the status of the order and providing a better UI to allow the user to have more functionalities which are already available in the api's.

7.Bibliography

- Service Oriented Architecture
  - https://en.wikipedia.org/wiki/Service-oriented_architecture
- Client-Server Architecture
  - https://en.wikipedia.org/wiki/Client%E2%80%93server_model
- Event-driven Architecture
  - https://en.wikipedia.org/wiki/Event-driven_architecture
- Microservices Architecture
  - https://en.wikipedia.org/wiki/Microservices
- Message Queue Benefits - AWS
  - https://aws.amazon.com/message-queue/benefits/
- Page of the curricular unit - TDIN
  - https://paginas.fe.up.pt/~apm/TDIN/
- NET Documentation
  - https://docs.microsoft.com/en-us/dotnet/standard/
- Restsharp
  - http://restsharp.org/
- Reactjs
  - https://reactjs.org/
- Docker
  - https://www.docker.com/
- Docker-compose
  - https://docs.docker.com/compose/overview/
- Nginx - Reverse Proxy
  - https://www.nginx.com/resources/glossary/reverse-proxy-server/
- RabbitMQ - javascript tutorial
  - https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html
- Pusher - javascript tutorial
  - https://pusher.com/docs/channels/getting_started/javascript
- Expressjs - API
  - https://expressjs.com/en/4x/api.html
- Postgres Documentation
  - https://www.postgresql.org/docs/11/index.html
- Mailtrap.io
  - https://mailtrap.io/

8. Resources

8.1. Software Used

- *Rider* provided by *Jetbrains*
- *Visual Studio 2019* provided by *Microsoft*
- *Visual Studio Code* provided by *Microsoft*
- *Intellij IDEA* provided by *Jetbrains*
- *Docker and docker-compose*

8.2. Setup Project

8.2.1. Installing Docker and Docker Compose

Before starting you'll need to have **Docker** and **Docker Compose** installed on your PC. The official instructions are in Install Docker and in Install Docker Compose.

Note: If you are getting permission error on the docker run hello-world or if you get a warning ".docker/config.json: permission denied run..." follow these instructions.

8.2.2. Configured containers

To start the environment :

$ docker-compose up

Note: To interact with the containers see *docker-compose.yml* and what ports are exposed and see also the *configs/nginx.conf*.

8.2.3. Use GUIs

Run the executable files accordingly the store or warehouse GUI.