Rotaract E-shop

Project Report

Elena Smeu – 253783

Adamantions Alexandros Kounis – 253711

Catalin-Andrei Tofan – 239984

Jakob Knop Rasmussen – Supervisor

Number of characters: 192,770

Software Technology Engineering
7th Semester
4th of June 2020

Table of Contents

Table of figures and tables

Abstract

The main drive behind this project is to offer a not-for-profit organization the chance to benefit from the digitalization technological solutions through the implementation of an e-commerce web application. From the stakeholder's perspective (in this case: ERIC Rotaract Europe), the most important objective is that the solution will morph their push supply chain strategy into a pull strategy whereas the developing team received freedom of choice to design and implement the features required to achieve that but also to expand that scope with supplementary aspects. Therefore, from a technical standpoint, the application is developed with the technologies that the team considered to have the most experience using for each purpose: frontend – Angular, backend – C# / ASP .NET with RESTful APIs and database – the Microsoft SQL management system. The main argument behind these technical choices is related to having a quality software that meets all the high priority requirements delivered in time with all the necessary documentation provided. The result is the Rotaract E-shop web application customized with the addition of the donate functionality for the organization's programs and where the main users: customer and administrator have differentiated access to functionality depending on their roles. In a nutshell, through this platform, the administrator can get an overview of and manipulate products, orders and donations whilst the customer can see donations and products items on the UI and interact with these through specific functionality (ultimately, being able to place an order and receive updates upon any progress made with them).

1 Introduction

Discussed also in the background description, e-commerce has the potential to offer new opportunities of growth and "increase efficiency in transaction management, including with respect to inventory, supply chain management and customer service"(E-commerce and Models, 2019) for companies and organizations alike. This report will explore and will offer more details on how such an e-commerce web application (referred to as the "Rotaract E-shop") was implemented for the stakeholders: ERIC. The European Rotaract Information Center is the actual Board that coordinates the activity of Rotaract Europe which is an international non-profit organization that activates in 47 European countries with a total of 116 districts and more than 20000 members. (*Rotaract Europe – Since 1988*) The organization aims to offer its members a platform through which they can develop leadership and cooperation skills by working on actions and events with impact on communities – both locally and internationally.

The main challenge addressed by this project is to implement an e-commerce software solution that will facilitate the transition from a push supply chain strategy to a pull one for ERIC's official merchandise which are usually purchased by clubs and members around Europe for various reasons such as: events, campaigns or PR. Currently, ERIC depicted the situation as one which leaves them the only option of building stocks of products in anticipation of members requesting them by email or phone which is unreliable, expensive and requires a lot of storage space. Besides the fundamental objective, the implementation of such system also aimed to offer additional functionality that will provide new opportunities for the board of Rotaract Europe and were not necessarily required by the stakeholders such as: management features for administrators, a personalized interface or the idea to integrate donations for various programs that Rotaract currently has or might develop in the future.

This report will offer insights into how most relevant requirements and features were defined and completed, the programming strategies and tools that made that possible and the way the outcomes were tested after their implementation was concluded. Nonetheless, it must be stated that few delimitation were made during the initial phase of the project: in the case of the payment and invoicing systems, the arguments were related to time and resources constraints given the fact that a 3rd party system had to be integrated whereas the deployment to the ERIC's provided server will occur outside the scope of this project. In the case of offering more options for language or currency, it was considered that they are not crucial to the system as for now and they were not requested by the stakeholders either.

Finally, it is worth mentioning that this report is aiming to thoroughly cover aspects from all the relevant phases of the building cycle of the final software solution. Consequently, the upcoming sections of this report will introduce the reader to how the requirements were defined followed by a comprehensive synopsis on how the web application was analyzed, designed, implemented, and tested. The report will focus on four main Use Cases that are most relevant for detailed illustration purposes: "Register as new user", "Place an order", "Manage donations" and "Manage orders" since they include all actors and most layers and entities of the system. Additionally, the last sections of the report will state the conclusions reached at the end of the project and the prospects of the Rotaract E-shop application.

2 Analysis

This section of the project report reviews the most important constituent parts of the Analysis and regards the usage of diagrams such as: Use Case and Activity diagrams to describe some features and actors of the web application, their consistency with regards to the requirements and how these come into play in the quest of elaborating on the background description. This analysis concludes with outlining the Domain Model diagram which helped the developing team move forward into the Design phase of this project.

2.1 Requirements

2.1.1 Functional Requirements

The following functional requirements are extracted from the Software Requirements Specification document (*Appendix B*) as the most important to have features for the Rotaract E-shop application. Thus, they are equivalent to *Must, Should and Could* have from the **MoSCoW** prioritization technique:

- The system ***must*** provide the viewer with the possibility to register as a customer user.
- The system ***must*** provide the registered user with the possibility to log in the system and log out of the system.
- The system ***must*** allow the administrator to add a product into the catalog and delete a product from the catalog of Rotaract merchandise (as part of Administration page functionalities).
- The system ***must*** allow the administrator to add and cancel a donation program (as part of Administration page functionalities).
- The system ***must*** provide the viewer and customer with the possibility to browse through a catalog of Rotaract merchandise on the main page of the application.
- The system ***must*** provide the viewer and customer with the possibility to view more details about a product on a separate page.
- The system ***must*** provide the viewer and the customer with the possibility to view all the active donation programs on the main page of the application.
- The system ***must*** allow the customer to add products to the cart.
- The system ***must*** provide the customer with the possibility to see all the products added in the cart and remove products from the cart.
- The system ***must*** provide the customer with the possibility to place an order with the items stored in the cart.
- The system ***must*** provide the customer with the possibility to get an overview of all the orders placed in the system and their status.
- The system ***must*** provide the administrator with the possibility to get an overview of all the orders from the system and to change the status of the active orders.
- The system ***should*** provide the administrator with the possibility to edit a product and a donation program as part of the Administrator page functionality.
- The system ***should*** provide the customer with the possibility to donate to an active donation program presented on the main page of the application.
- The system ***should*** provide the administrator with the possibility to update the inventory of a product.

- The system ***could*** provide the registered user with the possibility to recover access to the account in the case of forgetting the password.
- The system ***could*** allow the customer to add a product to a wish list.

2.1.2 Non-Functional Requirements

This section describes the non-functional requirements for the web application which are divided into more sections as following:

Performance requirements:
The system must be available at least 95% of the time with its functionality being used by an approximated 5000 different registered users from 47 countries and 116 districts across Europe.
The initial version of the web application should be able to support up to 500 concurrent users.

Response time:
The successful registration of a new user in the system must not exceed 3 seconds to be processed.
At least 95% of the data updates from the system to the database and back to the UI should be done in less than 3 seconds.
The system must be able to load the content of the home page in up to 3 seconds after the user logs in.

2.2 Use case modelling

Following the Inception and Elaboration phases of the AUP methodology, a set of prioritized requirements was defined of which implementation would ultimately shape the final version of the Rotaract E-shop web application. The importance of these requirements is also reflected throughout analysis considering that the developing team had a starting point from which to define the features that will be integrated within the app and the kind of users that will be able to use them. Consequently, modelling the use cases for the application depended heavily on the previous steps and can thus be considered not only an elemental part of the analysis but also a reliable model for how the e-shop was implemented.

Presented below is the final version of the Use Case model which is aligned with the application's functionalities at the end of the Implementation phase. Despite the efforts put into the early analysis by the developing team, it was inherent that the use case will suffer modifications due to decisions made regarding its simplification to a version that mirrors the actual implementation, various mistakes identified at a later stage or updates from stakeholders/supervisor. Therefore, the following diagram can be compared with an earlier one which can be found in the *Appendix C* for consistency check.

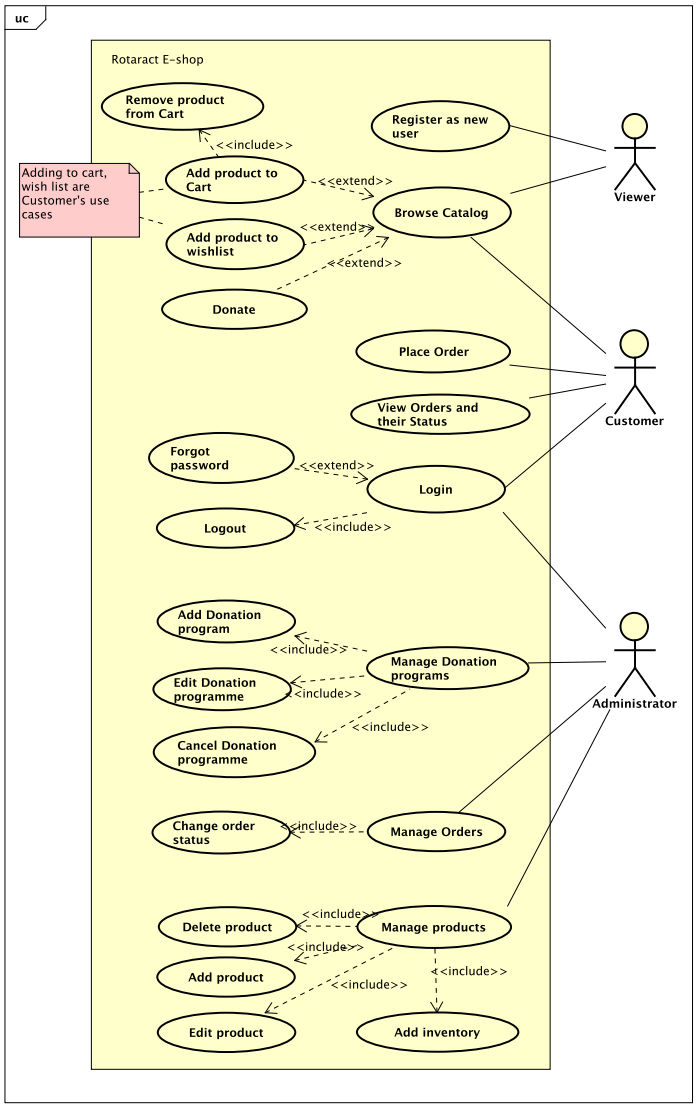


Figure 1 Use Case Diagram

Well stated also in a document cautiously studied by the team: "It is useful to bear in mind that previously written use cases and contracts are only a guess of what must be achieved. The history of software development is one of invariably discovering that the requirements are not perfect or have changed." (Larman, 2001) However (and an aspect that can be verified by comparing the versions), the developing team avoided to make major changes to the initial analysis such as altering the must-have functionalities as determined by the requirements and this decision was intrinsic to the manner in which the team based the implementation of the application on a logical flow from early analysis towards the late stages of the implementation with no major disruptions occurring in-between.

As it can also be seen from the above use case, the Rotaract E-shop application has common and differentiated functionality depending on the role of the user.

The main actors of the system are the *Viewer*, the *Customer*, and the *Administrator* whilst most part of the implementation revolves around the customer and the administrator as the main roles for the shop.

The viewer is regarded as a pseudo-actor by the developing team of the system mainly to differentiate between the registered and the unregistered user with the later having only limited access to functionality within the system. Thus, the unregistered user can view the Home page ("Catalogue page"), can view more details about products or donations and can also donate for a chosen donation program available on the home page. On the other hand, whenever such user registers an account, he/she becomes a customer of the Rotaract E-shop and gains access to more functionality such as: adding/removing products from cart, placing an order with the cart items or viewing and receiving details about the order progress. In the case of the administrator, initially the developing team designed the use case "Edit customer role" to let an administrator delegate the same role to a customer – use case which was removed after agreeing with the stakeholders that special and limited accounts will be used for this role and they will be attributed to chosen ERIC members as part of their internal transition processes.

In terms of functionality, the administrator holds an overview of all products, orders and donation programs from the system and can manipulate them: add, delete, edit in ways for which more details will be provided in the upcoming sections. Additionally, the administrator oversees the inventory, thus making sure that there is (and if there is) enough supply for the customer's orders to be processed and confirmed. This corresponds to a special demand from the stakeholders which will offer them more control in making a steady transition between the push and the pull supply chain strategies: the core challenge to which this software solution tries to respond and correspond.

2.2.1 Activity diagrams and Use Case descriptions

Another important integrated part of the early analytical process was to define a logical flow between the interaction of the user with the UI and the response of the system back to the user that could at least set the basis for the later implementation decisions. Despite suffering minor changes over the course of the project lifecycle, the activity diagrams designed in the Elaboration phase (with their subsequent descriptions) provided valuable insights and allowed the team to progress and to easier communicate the intentions with the stakeholders.
Presented below are few of these activity diagrams, the rest of them being available for further thoroughgoing study in the *Appendix C* of this project:

2.2.1.1 Register as new user

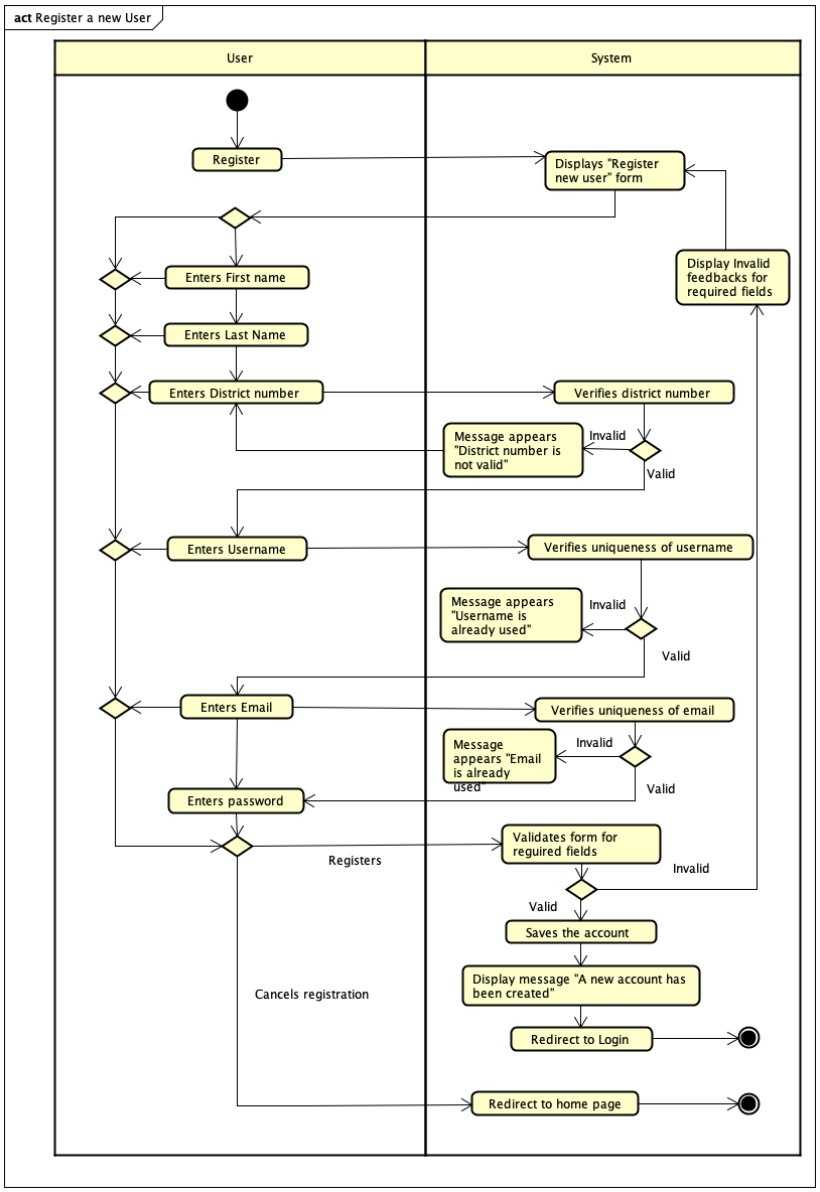


Figure 2 Register as new user Activity Diagram

UseCase	Register as new user
Summary	A viewer that wishes to make a purachase/donation has to be a user to do so. The registration paged is used for any rotaract club member to create their personal account into the system.
Actor	Viewer
Precondition	The Viewer has to be part of a rotaract club if he/she wants to register and therefore be in a possesion of a district number for his/her respected club, which is required to register into the system. The Viewer must have a valid and unique email address that can be accessed.
Postcondition	A new account is created with the specified email address and password, and is stored in the database.
Base Sequence	1.The viewer enters the First name. 2.The viewer enters Last name. 3.The viewer enters a valid district number. 4.The system verifies the district number. 5.The viewer enters the username. 6.The system verifies the username. 7.The viewer enters the email. 8.The system verifies the email address. 9.The viewer enters the desired password. 10.The viewer registers the account. 11.The system redirects the register user for login.
Branch Sequence	
Exception Sequence	Step 1-3 same as base sequence 5. The system shows an error message saying "District number is not valid." Step 1-4 same as base sequence 6.The system shows an error message saying "Username is already used." Step 1-7 same as base sequence 8.The system shows an error message saying: "Email address is already used."
Sub UseCase	
Note	The registration can be canceled at any time and the viewer is redirected to the home page.

Figure 3 Register as new user UseCase Description

2.2.1.2 Manage Donation Programs

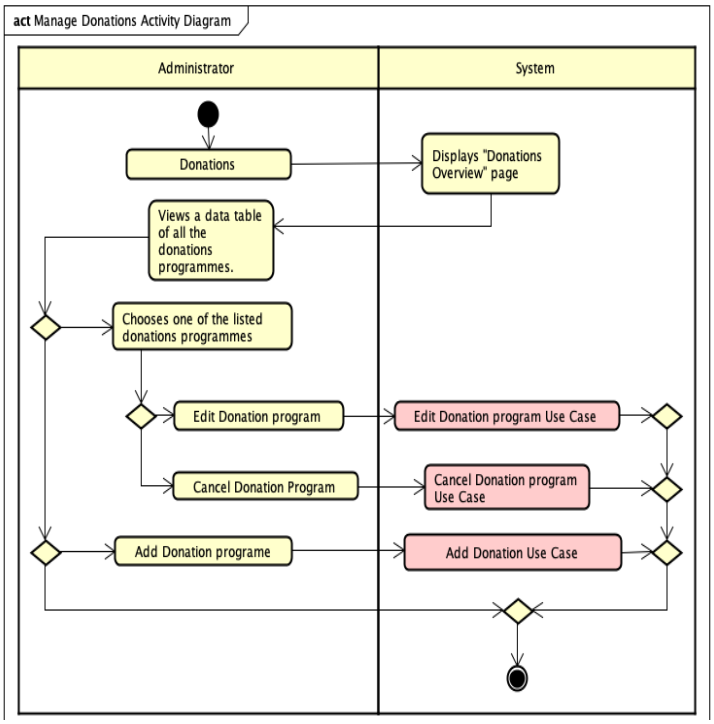


Figure 4 Manage Donation Programs Activity Diagram

UseCase	Manage Donation programs
Summary	The Administrator can view the donation programs already stored in the system and add a new program, delete or edit the existing ones.
Actor	Administrator
Precondition	The user is logged in the system.
Postcondition	The administrator was able to add, delete, edit or view a donation program in the system.
Base Sequence	1. The administrator clicks "Donations" button on the Administrator menu. 2. The system displays an overview of the donation programs already stored. 3. The administrator navigates the data table using pagination if necessary. 4. The administrator chooses a donation program listed in the table. 5. The administrator chooses to add a new donation program. 6. Use Case Add Donation Program starts.
Branch Sequence	4.a.1 The administrator chooses to cancel the donation program. 4.a.2 Use Case Cancel Donation program starts. 4.b.1 The administrator chooses to edit the donation program. 4.b.2 Use Case Edit Donation program starts.
Exception Sequence	
Sub UseCase	Add Donation program, Edit Donation programme, Cancel Donation programme

Figure 5 Manage Donation Programs UseCase Description

2.2.1.3 Manage Orders

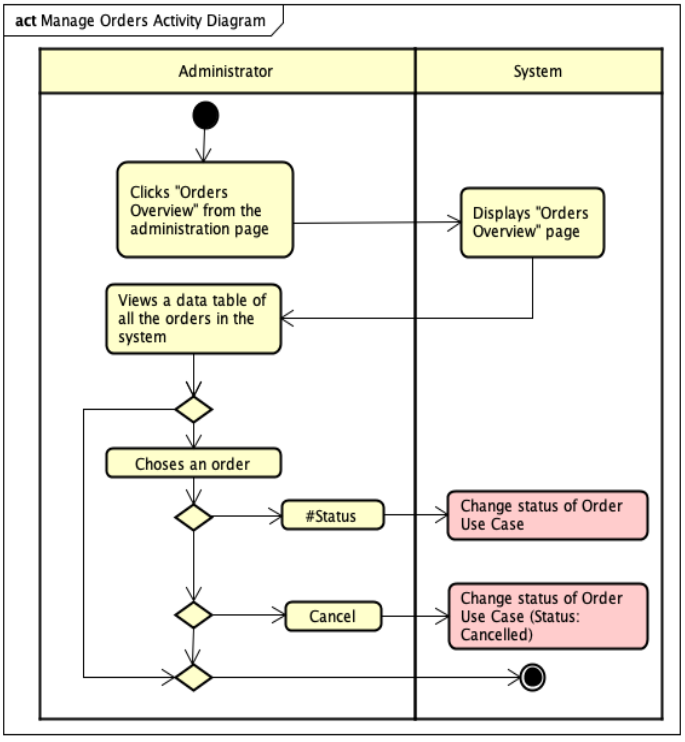


Figure 7 Manage Orders Activity Diagram

UseCase	Manage Orders
Summary	The adminstrator is able to view both the active and the archived orders from the sytem, to change the status of an order and view more details about it.
Actor	Administrator
Precondition	The administrator is logged into the system.
Postcondition	The administrator was able to view the orders made by customers and update their status. The system was able to save the updates and to notify the customers depending on the case.
Base Sequence	1. The administrator chooses "Orders overview" from the administration menu. 3. The administrator chooses between orders status tabs. 2. The system displays a data table list with all the orders stored in the system with the specific status. 4. The administrator chooses one of the orders. 8. The administrator chooses a status.
Branch Sequence	4.a.1 The administrator chooses to cancel the order. 4.a.2 Use Case Change order status (to 'CANCELLED') starts. 4.b.1 The administrator chooses to change status of the order . 4.b.2 Use Case Change order status starts.
Exception Sequence	
Sub UseCase	Change order status

Figure 6 Manage Orders UseCase Description

2.2.1.4 Place an Order

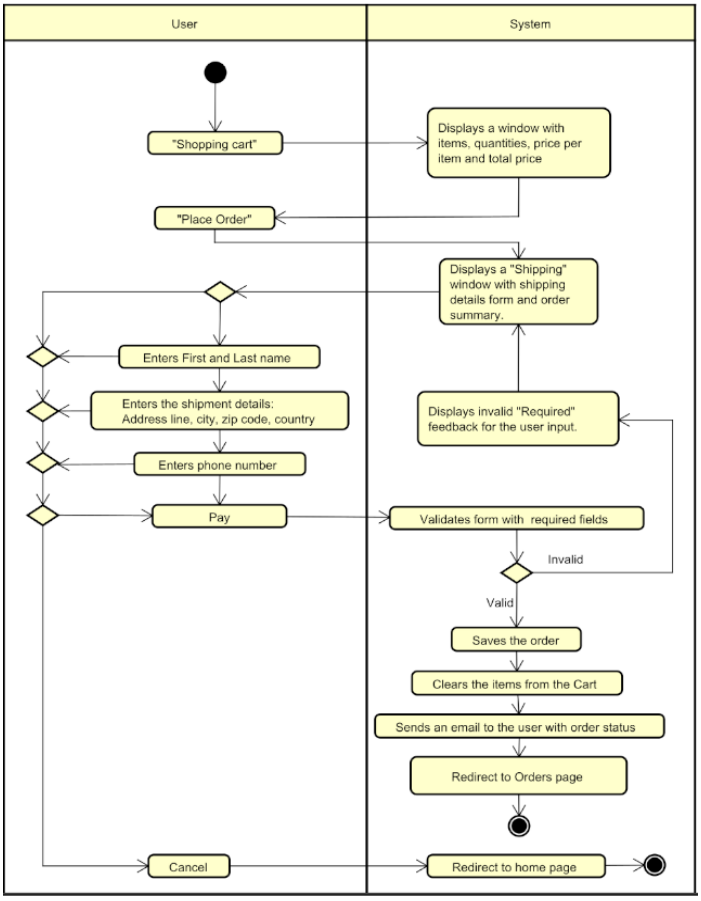


Figure 9 - Place an order Activity Diagram

UseCase	Place Order
Summary	The user is able to create an order with the items stored in the cart, add relevant purchase information and confirm the order.
Actor	Customer, Administrator
Precondition	The user must be registered, logged in the system and have items added for purchase in the cart.
Postcondition	The order has been successfully stored in the system.
Base Sequence	1. The user clicks on the "Shopping cart" icon. 2. The system displays the cart window with the items added so far and their details, quantities, price per item and the total price. 3. The user chooses to place the order. 4. The system displays a "Shipping detaills" page. 5. The user enters name, address and phone number on delivery address. 7. The user chosees to proceed with the payment. 8. The system checks if all the fields have been completed. 9. The system saves the order. 1 0. The system clears the items from the cart corresponding to that session. 11. The system generates an email that is sent to the user's email address regarding the order status. 12. The system directs the user to the Orders page.
Branch Sequence	7a. The user chooses to cancel. 7b. The system redirects the user to the home page.
Exception Sequence	
Sub UseCase	
Note	The payment process displayed in the Activity Diagram is not included in the UseCase description because of the project delimitation criteria.

Figure 8 - Place an order description

3 Domain Model

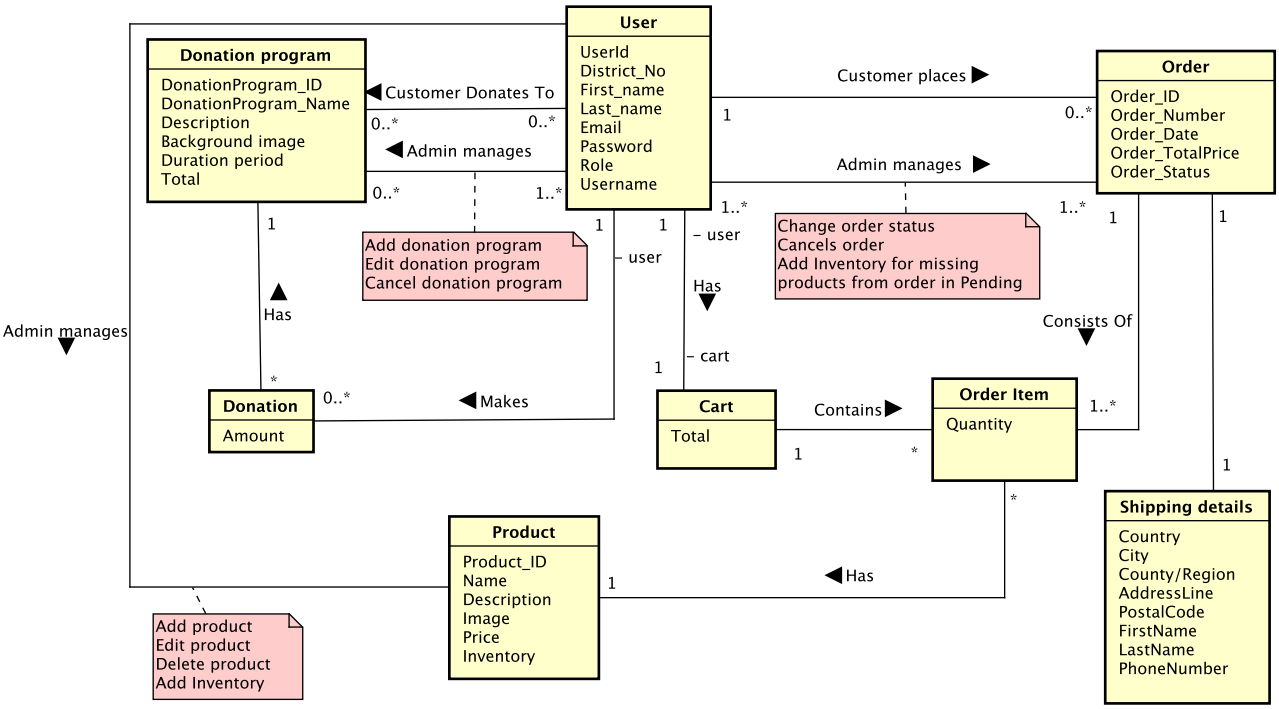


Figure 10 Domain Model

After the Analysis was to a great extent completed, a domain model of the system was made to illustrate the main entities and their interaction within the application which will further become the objects of the design model. These objects were further mapped into more descriptive diagrams such as class diagrams which mirror the actual implementation or sequence diagrams that depict the actions and messages exchanged by interacting software objects of different tiers.

The conceptual classes from Figure 10 are part of the Business Modelling component of the logical design. The diagram embodies dependencies, multiplicities and properties of these classes and further describes the Use Cases, therefore elaborating on the Problem Domain defined previously. The diagram features entity objects such as: User, Order, Product, Donation Program or Cart which are either taken out from the Use Case diagram or deduced from there as necessary for the system. As it will be described in the upcoming sections, these entities were transformed into data objects of either frontend, backend or database tiers of the system.

4 Design

This section of the report will guide the reader through different diagrams, patterns or technologies used by the developing team to serve different purposes and design-relevant

arguments for the choices made will accompany these figures. Besides this, the section will attempt to be the necessary preamble to the application's implementation details and to offer enough arguments for the consistency of this solution's constituent parts from artifacts of the Analysis to blueprints of the implementation.
"A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects. More precisely, a designer can describe the design of one or more scenarios of a use case; each of these is called a use-case realization." (Larman, 2001, Chapter 17). Taking this into consideration, the design section takes the same Use Case elements to describe their realization.

4.1 System Architecture

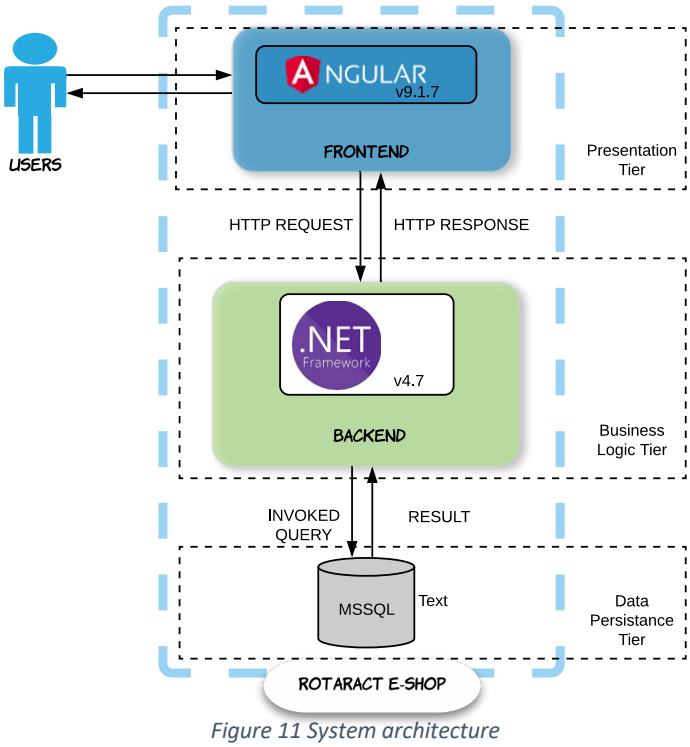


Figure 11 System architecture

Before any details on how the Domain Model was transformed into the Design Model for the Rotaract E-shop application, it is worth analyzing the underlying system architecture which also offers an introduction to the technologies used by the developing team while making the project.

4.1.1 RESTful Architecture

Representational State Transfer (REST) is an architectural pattern for distributed hypermedia systems. REST APIs enables the web application to have multiple possible CRUD (Create, Retrieve, Update and Delete) operations. The following HTTP methods were used and a brief definition is provided for each. (*What is REST – Learn to create timeless REST APIs*)

Table 1 - REST Methods

HTTP method	Purpose
GET	Retrieving data in JSON string format
PUT	Updating or modifying an existing resource, where a JSON string representation of the modified resource is sent in the request body
DELETE	Removing an existing resource by the Id, which is provided in the URI
POST	Adding new a resource, or for multiple purpose request, as it can produce different results for given the same input

Table 1 depicts one of the reasons why using RESTful resources was an assumed design choice: the benefit of having a uniform interface to transfer data integrated within the system. Additionally, the choice also considered that the Rotaract E-shop will make use of the stateless protocol for the client/server communication and that the objects will be manipulated from the URI.

4.1.2 Three Tier Architecture

The Rotaract E-shop application was designed on purpose to follow the Three Tier Architecture with the constituent tiers being described in Table 2. The choice for such architecture was made mainly to ensure that there is a clear separation of concern between tiers so that change on a tier will not impact other parts of the application. Besides that, this approach also ensured that the team could focus on areas of expertise during implementation and that the system can be scaled up and out instead of solely relying on a particular technology.

Table 2 - 3-Tier Architecture

Tier	Description
Presentation Tier	It consists of the user interface of the system, with which the user interacts through a desktop or mobile devices. In this case is represented by the Angular Web Application
Business Logic Tier	It consists of an ASP.Net Framework Web API application which holds endpoints that receive client requests, makes logical decisions, validations, evaluations and moves processed data between the surrounding layers.
Data Tier	It consists of a MSSQL database to which just the server can communicate with through an interface which is handled by the Entity Framework Driver.

(*5 Benefits of a 3-Tier Architecture - Izenda*)

4.2 Choice of technologies

This section describes most of the chosen technologies and tools used for the Rotaract E-shop and offers a brief argumentation on the reasons behind each choice.

4.2.1 Frontend Technologies

4.2.1.1 Angular

The motivation for selecting Angular for the Rotaract E-shop instead of alternatives is that Angular defines modularity within the component – architecture which results into more organized and scalable design. The main benefit outlined when this choice was made is related to the Angular-specific architecture of using "NgModules" as basic building blocks which provide compilation context for components. These modules were considered to offer a better approach of structuring the code for the pages of the application consistent also with the AUP process of multiple iterations (more specific having separate files for CSS, HTML and TypeScript inside each component). Therefore, one aspect of it was related to designing the application bearing in mind the reusability aspect. (*Angular - Introduction to Angular concepts*)

Choosing Angular was also related to how the system was initially defined to have differentiated functionality depending on the user. For the application to render faster, modules containing administration features should not load for the customer and vice-versa. Here, the solution was again provided by "NgModules" which include the "*lazy loading"* pattern that not only limit the amount of code required at start-up but also load components as well as modules on demand when certain functionality is accessed. In the latter case, the Dependency Injection (DI) pattern is used alongside to provide the application with classes and services whenever needed while Agular's Routing libraries would ensure a correct transition between the modules. (*Angular - Dependency injection in Angular*) Thus, Angular was a choice that would account for a more flexible, efficient, robust as well as testable and maintainable Rotaract E-shop application.

4.2.1.2 Bootstrap

"Bootstrap is a free and open-source CSS framework directed at responsive, mobile-first front-end web development. It contains CSS and JavaScript based design templates for typography, forms, buttons, navigation, and other interface components."(*Bootstrap (front-end framework) - Wikipedia*)
The reason for choosing Bootstrap for this application is related to its clean design and responsiveness provided by the grid layout. As for the integrated design elements (Angular widgets such as carousels, tab sets, date pickers, modals, tooltips and popovers) ng-bootstrap was chosen as a freely accessible library for angular components built using Bootstrap 4. (*Angular powered Bootstrap*)

4.2.2 Backend Technologies

4.2.2.1 ASP.NET Web API (.NET Framework 4.7)

Regarding how the system finally made use of the ASP .NET Web API with the .NET Framework 4.7 technologies for the backend side implementation requires more background information which includes the extraordinary circumstances under which this project was developed.

Initially, the Rotaract E-shop web application was intended to use the .NET Core with ASP .NET 5 in order to fulfill our intentions of developing a solution that will make use of the latest technologies available. Therefore, during the environment setup, this approach was the one chosen and a .NET Core application was created as framework for the backend logic. However, as the application began to shape up, an increased need of testing the APIs communication appeared which also increased the need of cooperation inside the team on the same matter.

Given the coronavirus extraordinary circumstances, this cooperation could only happen remotely when it was needed the most and therefore, the developing team decided to use Swagger to make all the necessary API tests remotely. However, the team encounter problems installing and configuring the "Swashbuckle.AspNetCore" package and, considering the importance of using this tool during the remote development of the application, the transition to the .NET Framework 4.7 was attempted and within the new setup, the configuration of Swagger succeeded. Considering that the framework was also a good alternative for the purpose of the implementation and that the differences when compared to .NET Core were rather insignificant, the team decided to proceed with the .NET Framework version of the backend implementation. (*Difference between .NET Framework 4.6, .Net Native and .Net Core - Stack Overflow*)

Regardless of that, the choice of using the .NET Framework for the server side of the application from a design point of view is related to some design principles that the Rotaract E-shop application aimed to integrate such as *interoperability* – since: "Microsoft ensures that older framework versions gel well with the latest version", *portability* – "Applications built on the .NET framework can be made to work on any Windows platform", *security* and the *simplified deployment* aspect. (*What is .NET Framework? Complete Architecture Tutorial*)

4.2.2.2 Swagger

Forward-thinking that the web application solution (especially with regards to the endpoints) must be designed in a way that makes it easier to document and test, the developing team considered relevant to use Swagger from a certain point during the development process. The point from which Swagger became an essential tool coincided with the above-mentioned circumstances and this can be arguably considered a milestone in the decision making process describing the development of the Rotaract E-shop application.
The choice also predicted that the Web APIs will more easily be implemented if there is a way to interactively visualize them and the request/response communication which would also ensure that potential errors will be captured earlier in this process.

4.2.2.3 Entity Framework

This framework was chosen for the backend side to act as a bridge between the actual domain classes and the database and the main reason behind making this choice was related to the idea of separation of concerns since the Entity Framework allows the developer to focus more on designing the model instead of the database used to store it.
"Entity Framework is a powerful ORM tool which is easy to understand and implement. With the help of EF, it is quite convenient to develop secure, scalable and robust enterprise software applications." (*Benefits of Entity Framework*) Designing the Rotaract E-shop emphasized less effort on database and storage without undermining their importance and therefore, Entity Framework was considered from the beginning a suitable solution which would simplify the mapping process from entities to raw data objects. Other benefits that were outlined when the technology was selected were the framework's simplification of queries through its LINQ

Provider (which could not be avoided during implementation) as well as the ease of scaling up the database schema by using migrations in the process.

4.2.2.4 MSSQL

"Microsoft SQL Server is a relational database management system (RDBMS) that supports a wide variety of transaction processing, business intelligence and analytics applications in corporate IT environments. Microsoft SQL Server is built on top of SQL, a standardized programming language that database administrators (DBAs) and other IT professionals use to manage databases and query the data they contain" (*What is Microsoft SQL Server? A definition from WhatIs.com*)
Using this technology resulted from a design choice made early in the development process: that the system will use a relational model to connect data elements to specific tables. "The relational model provides referential integrity and other integrity constraints to maintain data accuracy." (*What is Microsoft SQL Server? A definition from WhatIs.com*) Other reasons why this choice was made were related to avoiding redundantly stored data and to ensure the scalability and rapid growth of a schema that had no prior precise definition.

4.3 Design Patterns

4.3.1 Dependency Injection

Angular Dependency Injection framework implements the DI pattern. It is responsible of creating and maintaining the dependencies and to inject them in classes that request it.

The services or objects that are Injectable (marked with specific decorators) act as *Singletons* inside the Agular's DI framework. When a class is requesting a dependency, the Injector is either creating a new one – if it was not instantiated or return an instance that is already created inside the application. (*Introduction to Angular Dependency Injection - TekTutorialsHub*)

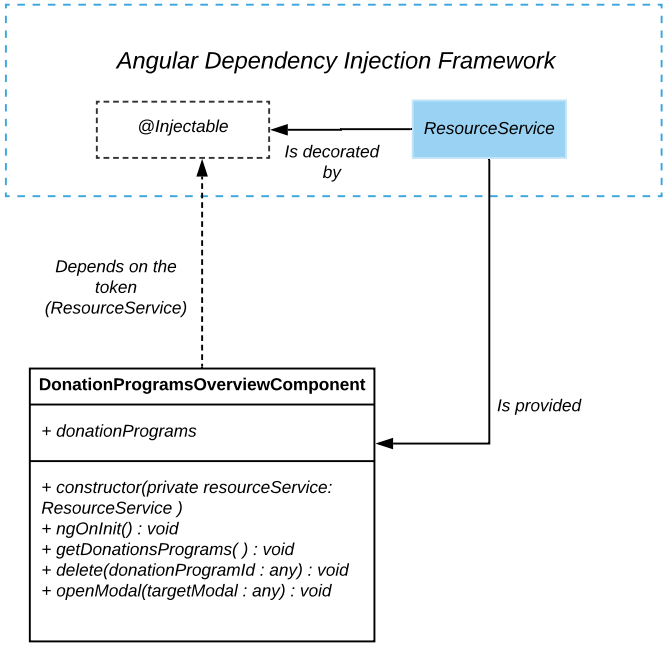


Figure 12 Dependency Injection pattern for Resource Service in Donation Programs Overview Component

Figure 12 describes how the "Resource Service" is provided to the "Donation Programs Overview" component by the DI framework. This design pattern was used for this application with the clear scope of reducing code's complexity by decoupling objects (components or services) from their dependencies. (*Inversion of Control Containers and the Dependency Injection pattern*)

4.3.2 Repository and Unit of Work

Switching from the frontend side to the backend, other design patterns of choice that served multiple purposes for the system are the Repository and Unit of Work patterns (which are intertwined to some extent). As described by the Microsoft documentation: "The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application." (*Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10) | Microsoft Docs*)

As it can be deduced from Figure 13, using these design patterns was a natural choice from having an ASP .NET application communicating with the Entity Framework and thereafter with the MSSQL and also an indirect consequence of using the RESTful architecture. Consequently, Figure 14 depicts how these diagrams were used for the realization of the "Manage Orders" Use Case and the two figures were added together to facilitate comparison. Additionally, these design patterns ensured an isolation of the application from changes in the data store and accounts for the testability aspect of the system for which more details will be offered on the Testing section of this report.

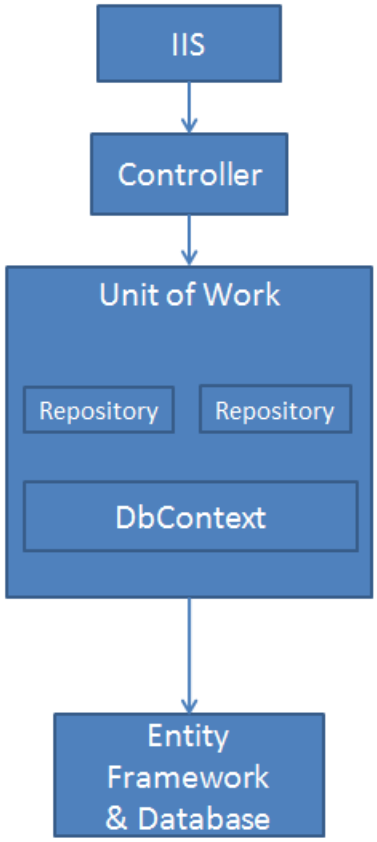


Figure 13 - Microsoft docs Unit of Work Diagram

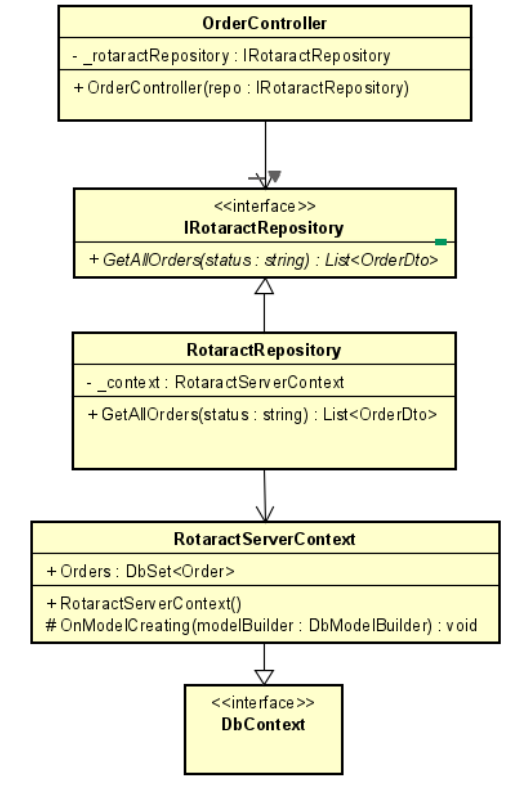


Figure 14 - Rotaract E-shop usage of Unit of Work pattern

4.4 Design Model Diagrams

In the previous sections of the Design, the Rotaract E-shop application was presented from the perspective of a few choices made regarding the underlying architecture, the tools and the

technologies selected to build the solution. This preamble was necessary to introduce the reader into the actual details for both design and its relevance for the implementation.

"A design class diagram (DCD) illustrates the specifications for software classes and interfaces in an application. In contrast to conceptual classes in the *Domain Model*, design classes in the DCDs show definitions for software classes rather than real-world concepts." (Larman, 2004) Constituent elements of the *Design Model* are in this project's case the class diagrams (more relevant for the backend) but also the package and routing diagrams of the presentation tier which altogether depict the system holistically. Consequently, these diagrams are the bridge between the *Domain Model*, *the artifacts of Analysis* and the actual implementation of the system.

4.4.1 Angular Diagrams

In this section, the presentation tier of the system will be described by providing the package (modules in Angular), API and routing diagrams for it. For a more detailed overview which includes services, helpers and more dependencies, the *Appendix D* is provided.

4.4.1.1 App Module Diagram

Figure 15 provides an overview of the main "*App Module*" and its nested modules and components. The diagram describes an organization of the modules driven by subsequent differentiated functionality for administration and customer (*catalogue* and *profile*) but also by shared functionality represented inside the *shared* and *auth* modules. Outlined with dependency arrows is the manner in which the modules and components are communicating with each other and an aspect worth mentioning here is related to the usage of *template binding* to bring functionality of a child component inside the parent component.
The structural approach of using Agular's feature modules offers the possibility to

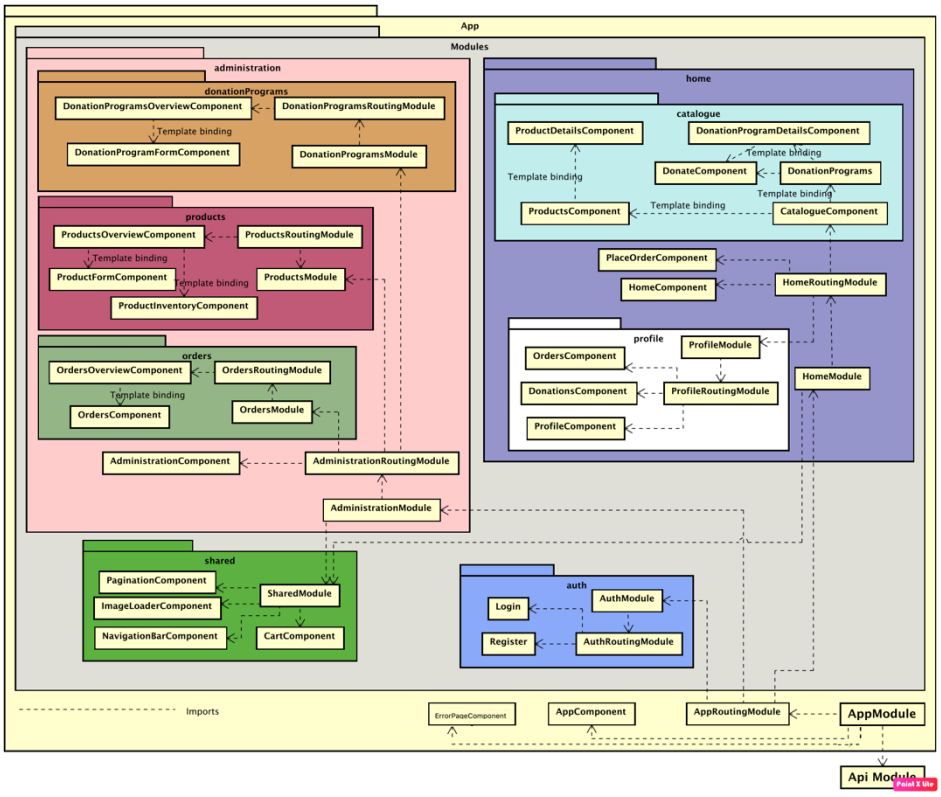


Figure 15 App Module Diagram Modules Diagram

further develop the design of the UI layer with more categories of modules, each bearing typical characteristics such as: Routed, Routing or Service modules. (*Angular - Types of feature modules*). Some of these categories are used inside the Rotaract E-shop application mainly due to the chosen approach of using *lazy loading* as a trigger method for imports and an overview of these is provided by the following table:

Table 3 - Feature modules categories

Routed Modules	AdministrationModule, HomeModule, OrdersModule, ProductsModule, DonationProgramModule, ProfileModule, AppModule
Routing Modules	AdministrationRoutingModule, HomeRoutingModule, OrdersRoutingModule, ProductsRoutingModule, DonationProgramRoutingModule, ProfileRoutingModule, AppRoutingModule
Widget Modules	SharedModule

Therefore, the main module of the angular application is importing its routing companion *AppRoutingModule*, which is using the *lazy-loading* method for importing the specific routed modules (AdministrationModule, HomeModule and AuthModule). The *AppModule* is further importing the *ApiModule* to provide access to the API implementation, for which more details will be provided in the implementation section. Besides this, the *AppModule* places the *AppComponent* as root component in the object's metadata used by Angular to create and render the host page.

4.4.1.2 API Module Diagram

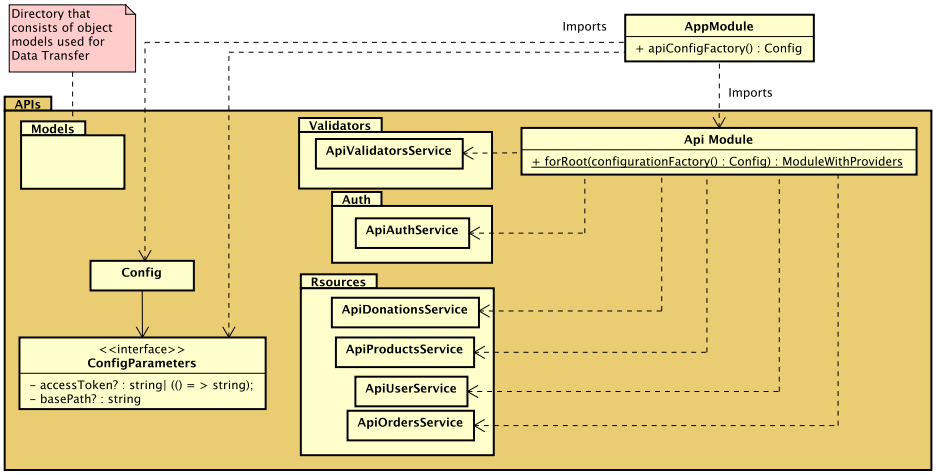


Figure 16 API Module Diagram

Figure 16 offers an overview of the *API module*, which is using the static method *forRoot* that has as parameter a *configurationFactory* method constructed in the *AppModule*. This returns a *Config Object* that holds the access token and the *basePath*. The object returned is further used

by the services inside the *ApiModule* for providing a RESTful layer of the communication with the server.

4.4.1.3 Routing Diagram

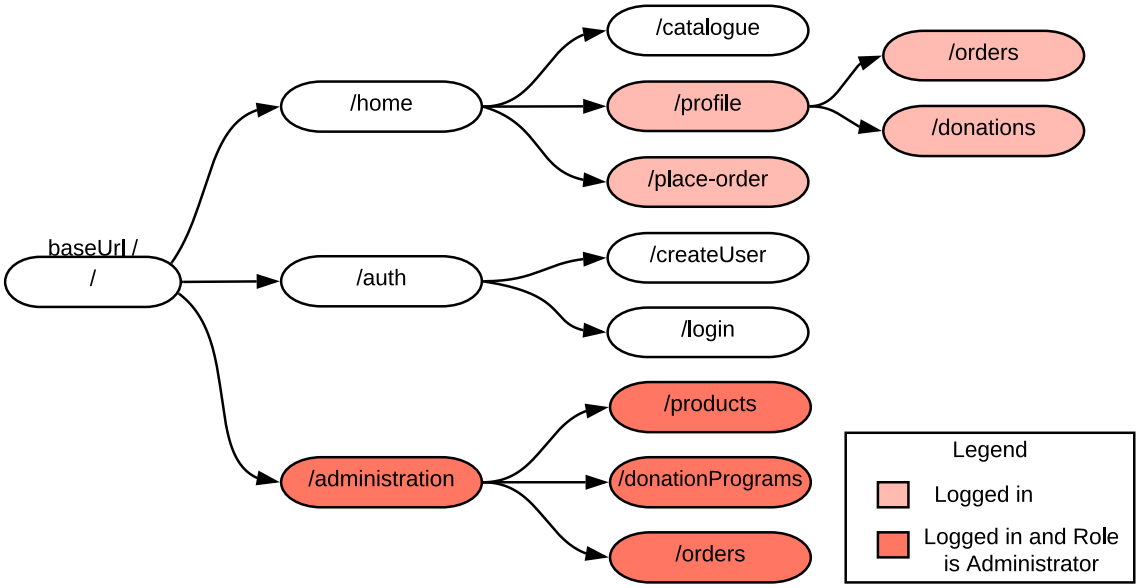


Figure 17 Routing Diagram Overview

Figure 17 is a snapshot of the Angular application routing design approach. The node routes are an URL image of the routing feature modules and the leaves represent components. The coloured routes are guarded by *AuthGuardService*. A detailed description will be provided in the *Implementation* section of the report.

4.4.2 Back-End Class Diagram

This section introduces the actual depiction of the Business tier logic and relationships in the form of a class diagram. Building the diagram was a result of multiple iterations following a *Code First approach* which scaled up the server side as required by the development process. (*Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10) | Microsoft Docs*) The back-end is based upon the skeleton of an ASP .NET MVC application (which however cannot be referenced as fully implemented MVC pattern since the *View* is the actual Angular application), it integrates the context of using Entity Framework with the Unit of Work pattern described earlier and the result of that. However, given the size of the diagram, the entire representation of it could not be included in the report as a figure and therefore Figure 18 displays only a reduced view of it without operation and most of attributes. The full class diagram can be analyzed in the *Appendix D* of this report:

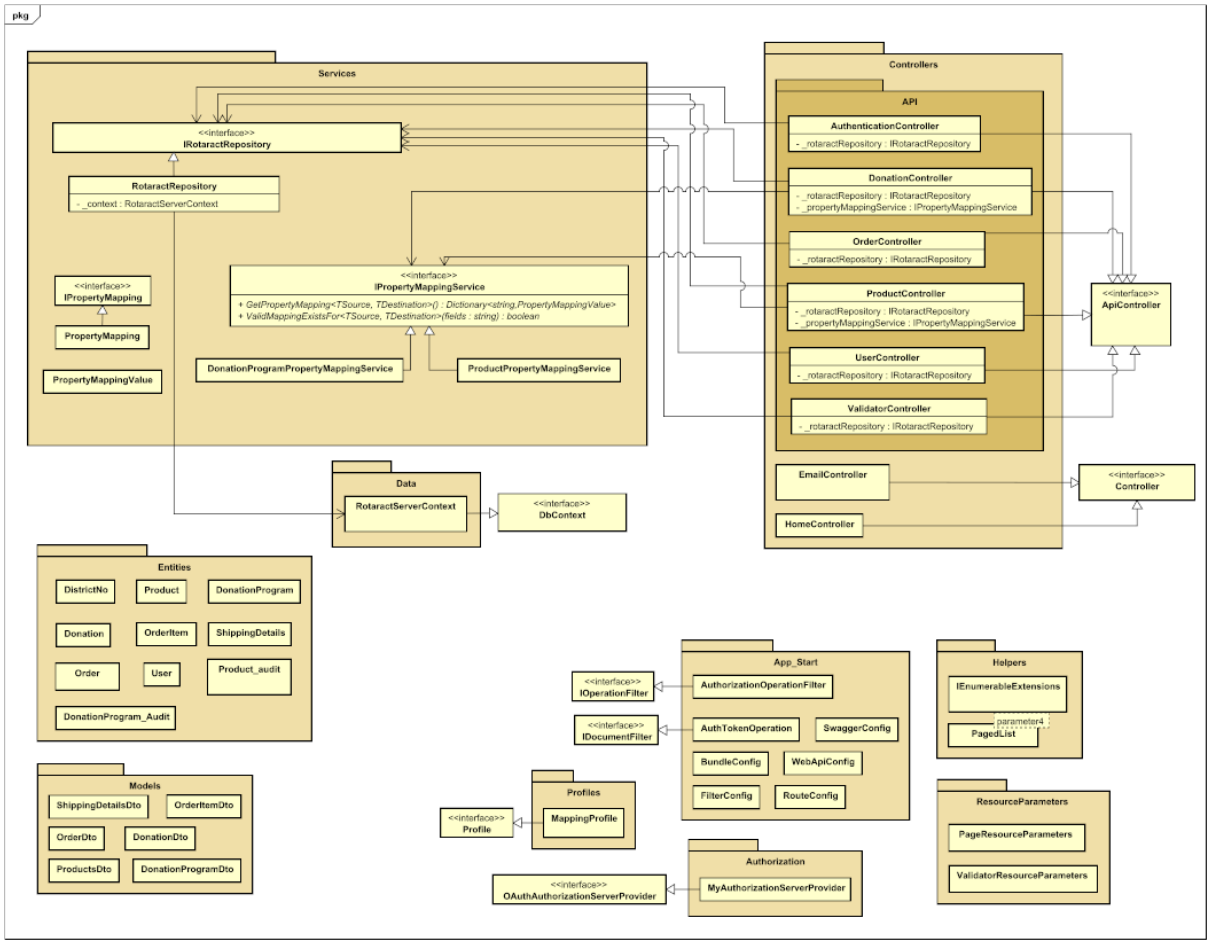


Figure 18 - Back-end Class Diagram

As the diagram displays, relevant classes for the implementation are grouped in packages such as *Models*, *Controllers* or *Services* for clarity reasons. Regardless of the *Helpers*, *Resources, Authorization* classes of which usage might be described in the *Implementation*, the diagram shows different dependencies and interface implementations that were required during development. The *Models* from the bottom left side of *Figure 18* are mostly relevant for the communication between the UI and the server as *data transfer objects* and their references are used as parameters for the *API Controller*'s methods which are defined to support the *Use Cases realization*. These Controllers are the server's APIs that facilitate the communication between tiers through *HttpResponseMessages* and they take a reference of the *IRotaractRepository* interface to ensure the level of abstraction and the implementation of the *Unit of Work* pattern as described in a previous section.

RotaractRepository class implements methods of the interface it connects with and takes a reference of the *RotaractServerContext* which on its turn implements the *DbContext* interface. This system specific context class coordinates Entity Framework functionality and specifies which entities are included in the data model. *RotaractServerContext* creates a *DbSet* property for each entity set. "In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table." (*Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10) | Microsoft Docs*) These entities that are transformed from raw database data to entity objects are depicted in the *Entities* package and some of them relevant also for the 4 use cases chosen initially are: *Order, User, Product* and *DonationProgram.* More details regarding how methods are implemented and how this transformation process occurs will be described in the *Implementation* section of the report.

4.5 Sequence Diagram

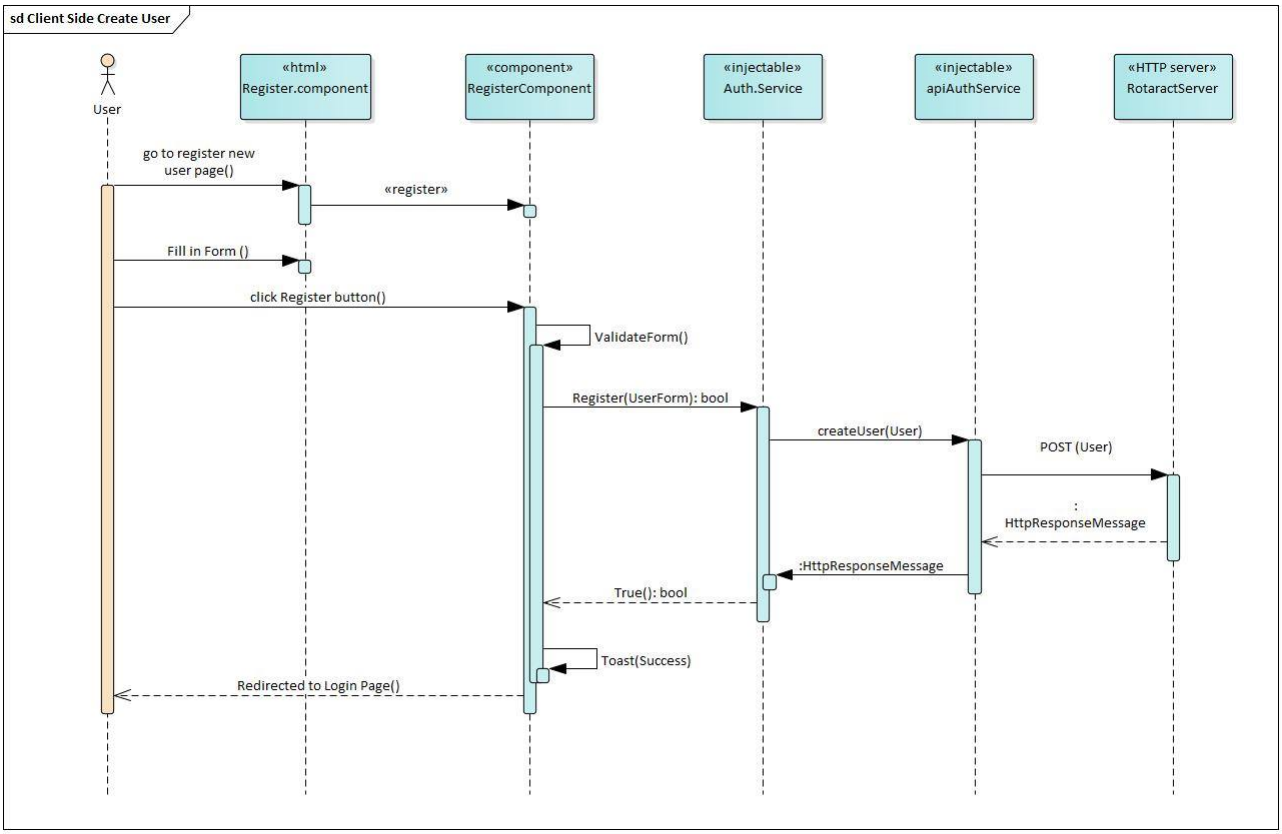


Figure 19 Sequence Diagram Register a new user - Angular application

Figures 19 and 20 illustrate the sequence diagram for registering a new user into the system. *Figure 19* describes the sequence of the *presentation tier* and *Figure 20* describes the sequence of the *business and data tiers*. To oversimplify the functionality and avoid the calls made to the server side about checking for the uniqueness of the username and the email address as well as the validity of the district number, in the sequence diagram those are being skipped.

Firstly, the user is directed to the register new user page which is calling the register component. The user fills in the form. After clicking the *register button*, an if statement ensures that all of the fields have been filled accurately. Thereafter, the sequence accesses the register method in *AuthService* while passing the *UserForm* which passes it further to the *apiAuthService's CreateUser Boolean* alongside with the User. Next the *Http Post* occurs and the sequence for that is presented in *Figure 20*. After the response of type *HttpResponseMessage* is received, a successful message in the form of a TOAST is displayed and the user is redirected to the login page in which he/she has now the valid credentials to login.

While the following description is what happens after the post request in the server side. The order of functions in the server side begins with the *AuthenticationController* being called. Despite already having validators for the *Username* and *email* uniqueness and district number validity the server is once again checking that information before registering a new user into the database. After having done with consecutive requests to the *RotaractServerContext* which extends to the database, it executes the *addUser* method from the *RotaractRepository* and then adding it to the database following that with a save statement and an *Ok status code* in the *HttpResponseMessage*. In the event of any of the conditions failing, the appropriate *Error Status code* is sent back to the client side.

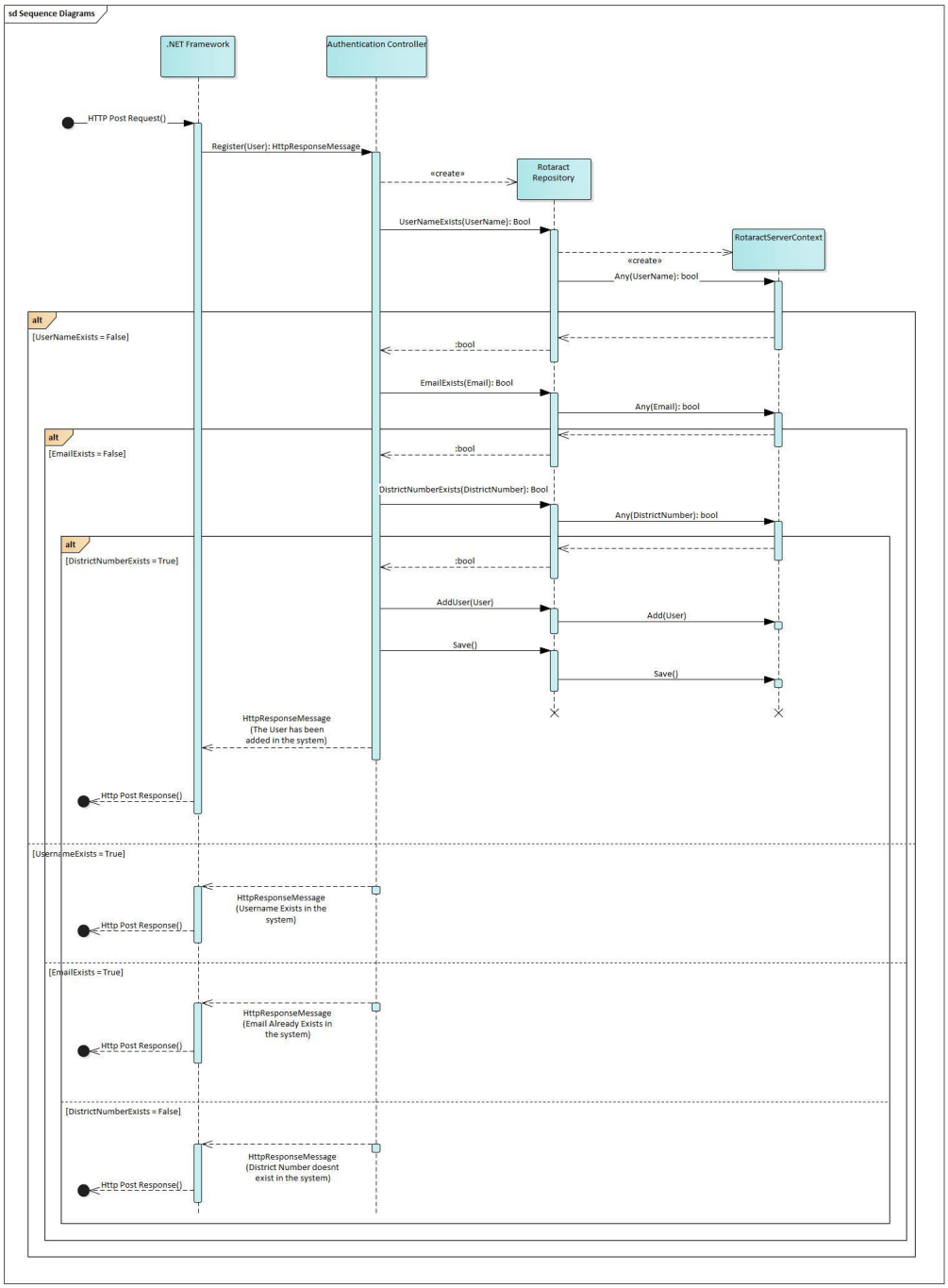


Figure 20 Sequence Diagram Register a new user - Server

4.6 UI Design

The UI of the Rotaract E-shop application was designed in a simple manner, using some of the bootstrap UI components and a limited number of external libraries like date pickers, modals and alerts. However, the light design is based on the white containers for content and the color choices are based on the Rotaract Europe branding libraries. More details related to the actual design implementation are provided in *Section 5.1.2.*
(*Brand Identity – Rotaract Europe*)

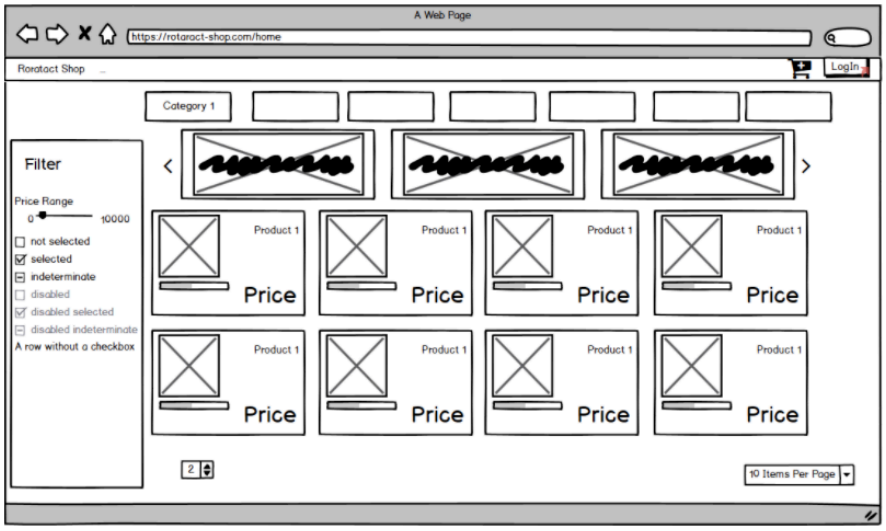


Figure 21 Catalogue Page Mockup

Figure 21 represents a snapshot of the earlier conceptual design of a catalogue page. The page in figure *21* (as it is the case with all of the displayed pages from this section) was created using *Balsamiq* as a mockup tool. The importance of having these earlier design representations, despite being used mostly as guidelines, is revealed also by the actual UI used for the Rotaract E-shop which included most of the elements from these mockups – such as, for the case of *Figure 21,* the carousel, containers or pagination elements that will be described later in this report. Therefore, not only did they provide insights about a possible realization of the *Design Model* to the stakeholders but also increased the speed of its later implementation by having a more UI related representation of the model that was different and more potent than what the early design model diagrams could offer.

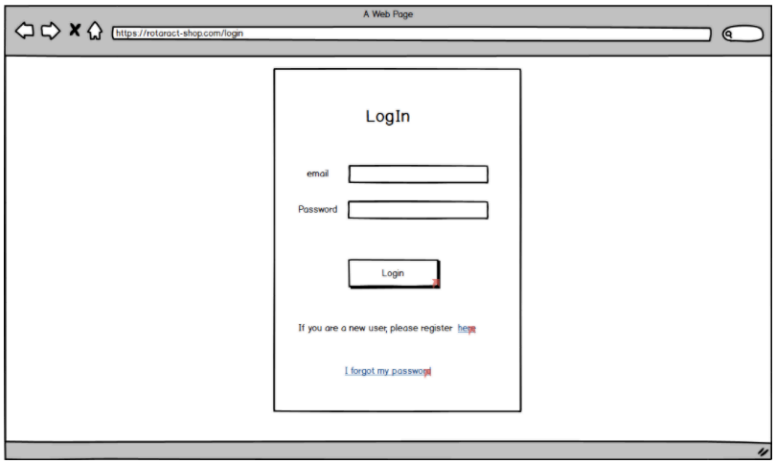


Figure 22 Login- Mockup

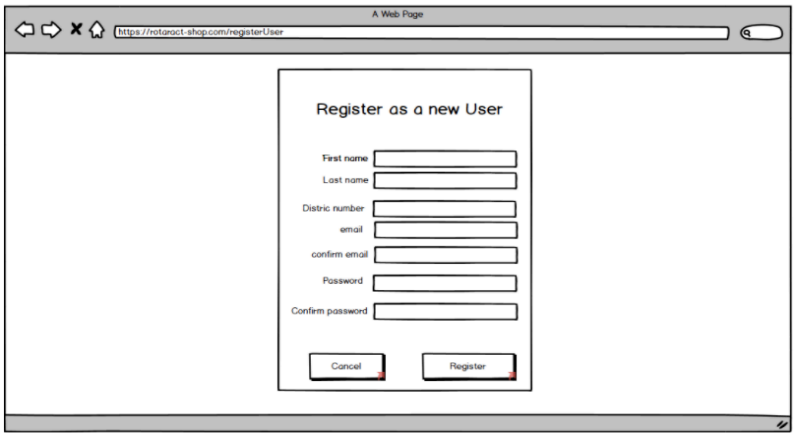


Figure 23 Register as new User Mockup

Figure 22 and *Figure 23* display the initial designs for the login and respectively for the register pages which proved relevant for the implemented UI through the fields that they contain. These pages also used the *Domain Model* as primary guideline for understanding what might be required from the user to input either in the form of credentials or additional information prior to the authentication. Both figures provided insights into what fields should be mapped as required and what kind of validation would likely be added to them (as it is the case with the *district number* that was required from any potential user of the application as a proof for being a member of a Rotaract Europe club prior to the registration).

Figure 24 illustrates the conceptual design of the catalogue page after the user logs in and, if compared to the implemented one from *5.1.2*, it can be again seen that they do not differ too much. This proves a level of consistency between what was projected immediately after the *Domain Model* was shaped but also defines elements that the implemented UI did not include such as the filtering functionality (as this was not even considered as a functional requirement but rather as a nice to have functionality outside the scope of this project).

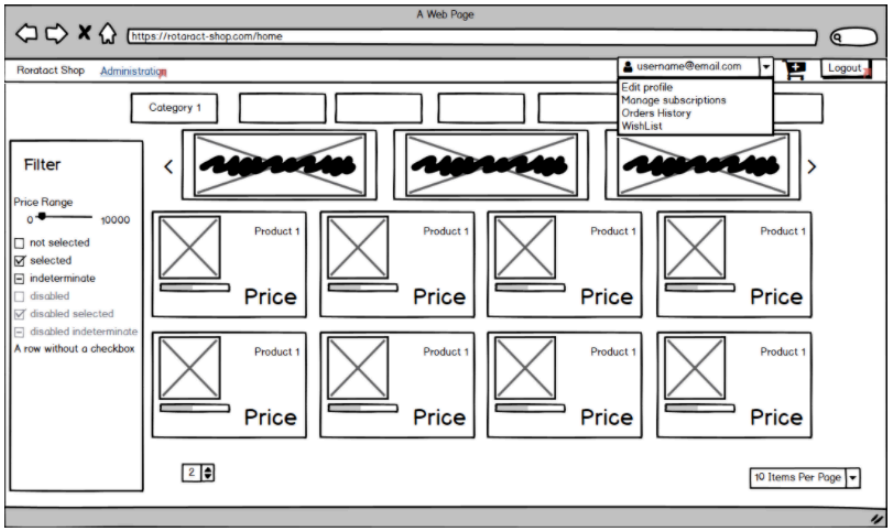


Figure 24 Catalogue page after login Mockup

Figure 25 is the mockup design that proved to be the most relevant for the implemented UI and which also includes Rotaract specific elements such as the logo. This initial concept was

inspired from other examples of e-commerce solutions but also contains elements from the *Domain Model* specific to this project.

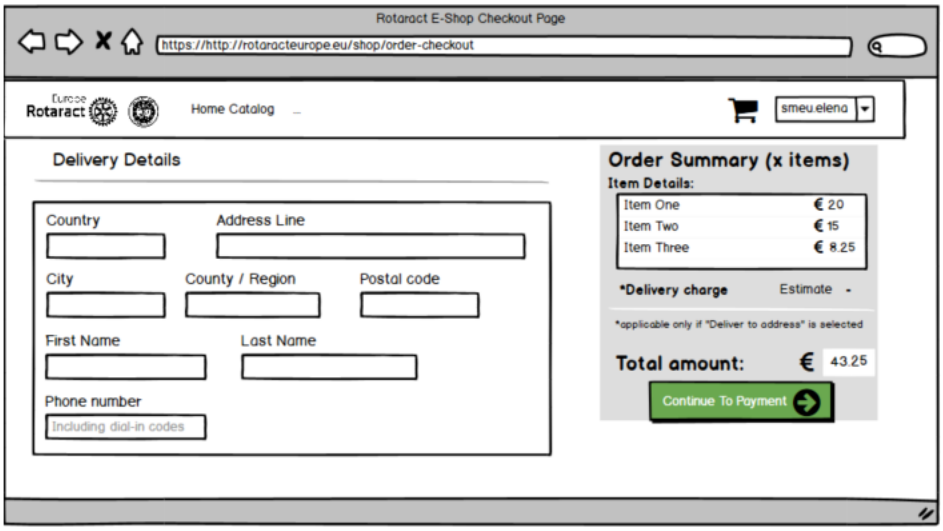


Figure 25 Place an order Mockup

Figure 26 illustrates the only mockup design that was made for the administration page and that also proved relevant for the actual design implementation. Its conceptual characteristic is proven by the presence of the *Users* menu which was initially considered needed for allowing the administrator to change the role of an existing user but was finally removed after a meeting with the stakeholders.

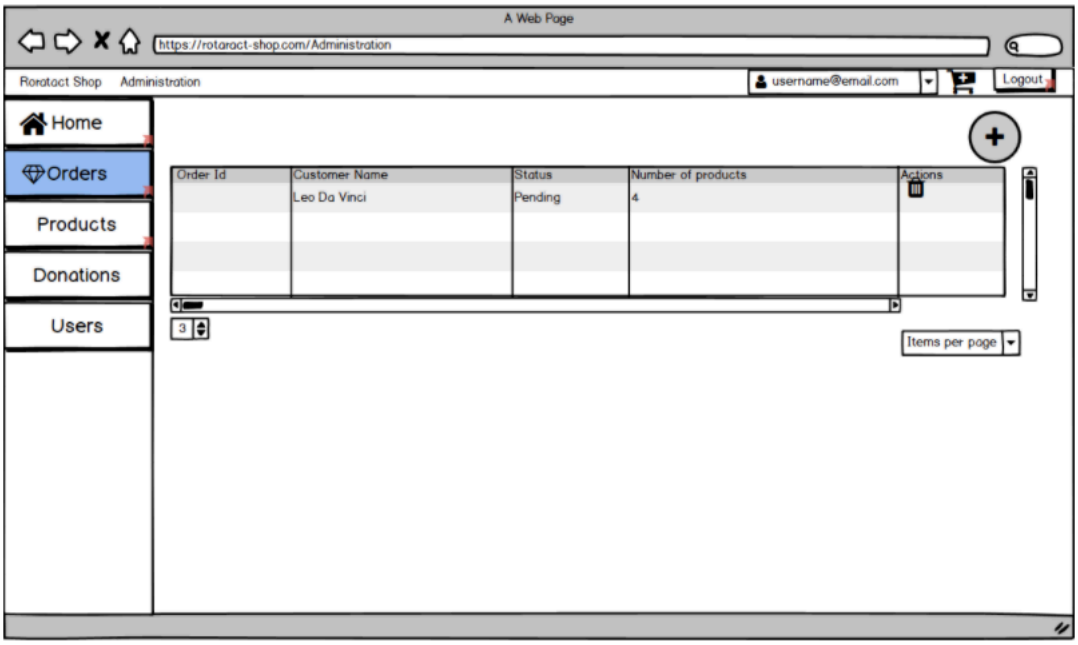


Figure 26 - Manage orders page Mockup

4.7 Database Diagram

Represented also in the server-side class diagram, the *ER diagram* from *Figure 27* mostly defines the database entity objects mapped as tables where entries for each entity are mapped as rows in the tables. These entity objects are the ones parsed to the *RotaractServerContext* as *DbSets* in order to become tables in the database and the ER diagram depicts all of the included properties (such as *OrderId* or *TotalPrice* for the *Orders* table), PKs or FKs for the relevant properties that define the relational feature of the MSSQL databases through the usage of constraints between tables as well as multiplicities between the entities (such as: *one user can have 0-to-many orders stored in the system*). These entities are updated and maintained through the basic persistence storage functions of computer programming: *create, read, update and delete (CRUD)* which translate and correspond to the POST, GET, PUT, DELETE methods of the RESTful APIs, enabling for the complete feedback of information between UI and the database to be established. More details regarding how this happens in practice will be provided in the upcoming sections of the report.

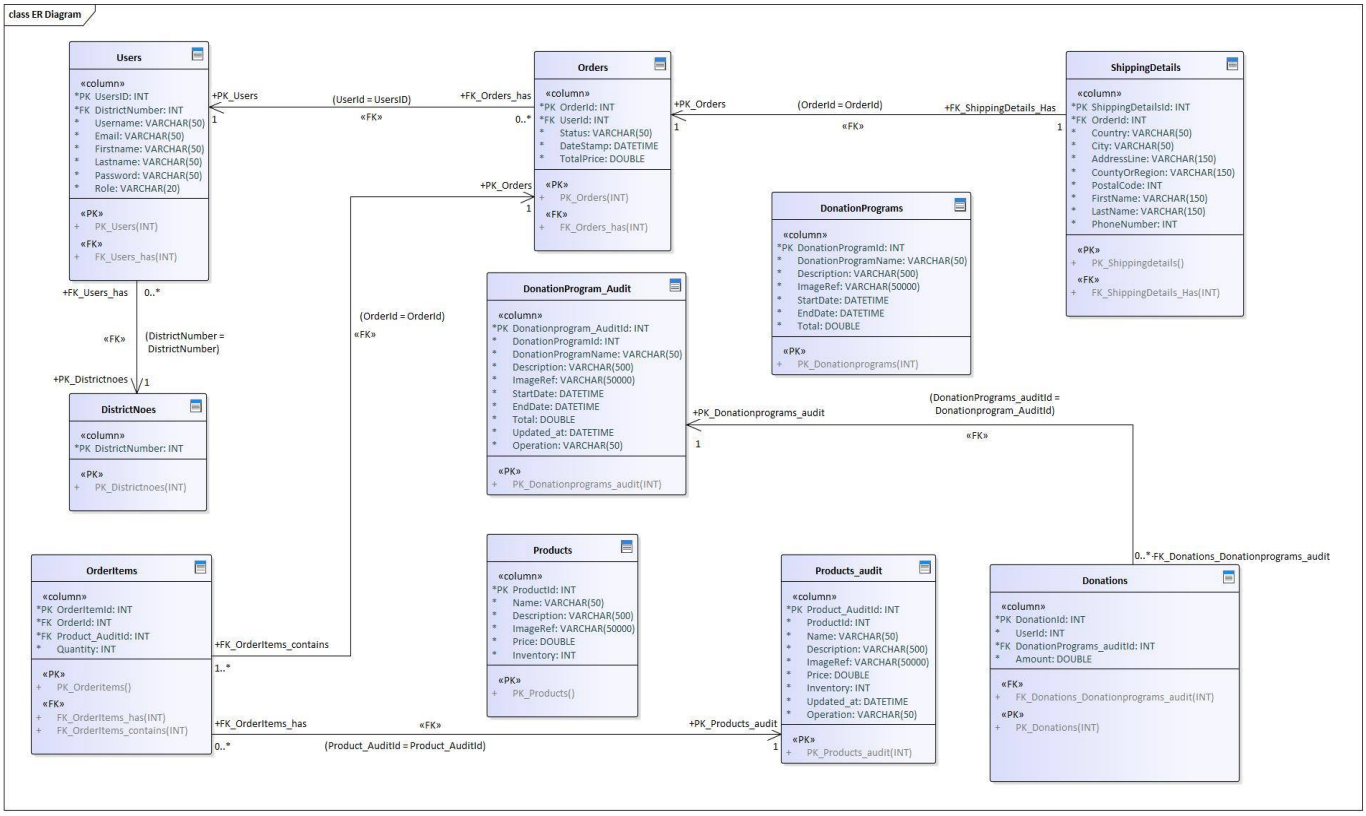


Figure 27 Rotaract E-shop ER Diagram

4.8 Deployment Diagram

The Deployment diagram shown in *Figure 28* is presenting the physical devices that the solution would normally use in a production environment and the interaction between them.

The diagram reflects a configuration that might not be definitive and which might include in alternative configurations a container such as *Docker* or a cloud service. However, as it can be seen in the diagram below, the server will contain the three-tiers of the system and a client will access this server using a TCP/HTTP connection.

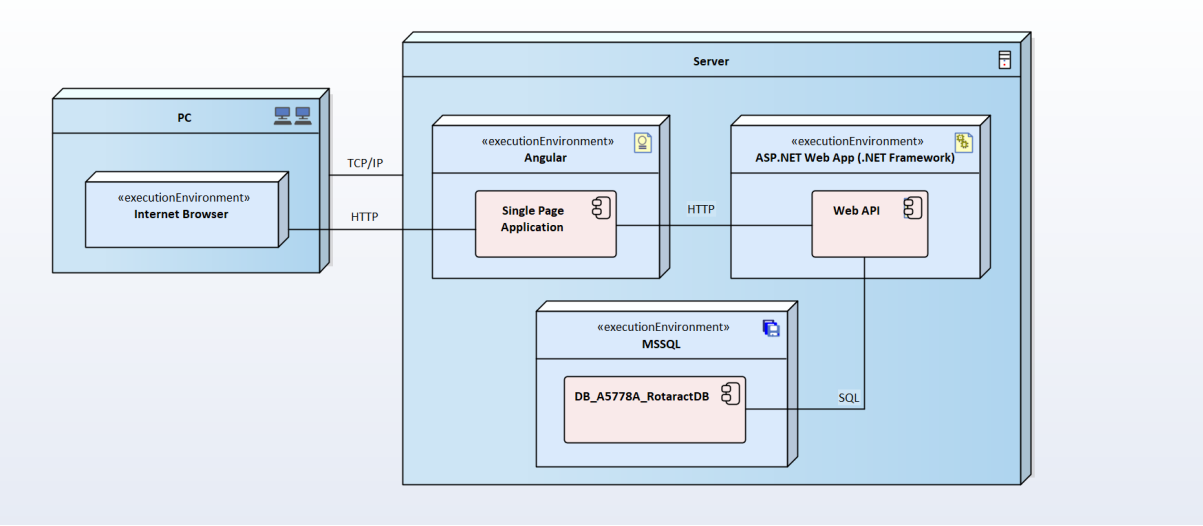


Figure 28 Deployment Diagram

5 Implementation

This section which describes the actual *Implementation Model* in AUP terminology is the one to offer the *HOW* with regards to the transformation of the analysis artifacts into objects of implementation. Therefore, the approach that this segment of the report will follow is aimed to be consistent with the initial one of observing the realization of the selected Use Cases. As it is the case here, this realization will be described with relevant segments of code supporting various methods, components or patterns that generate expected functionality as per the one described by the requirements. These elements will be taken from all tiers of the application and will aim to extract the essential aspects of their implementation.

5.1 Presentation Tier

As discussed before, this section's tier was implemented in the latest version of Angular using code written in CSS, HTML and TypeScript. The upcoming subsections aim to clarify interesting aspects throughout this tier's development and the realization of some main features.

5.1.1 Single-web page and Routing

Angular is using a component-based architecture for its single-page applications and a core highlighted benefit of that is the *lazy-loading* approach for routed feature modules and routing feature modules. This benefits the implementation by having a better optimized single web page with regards to the functionality specific for each user, thus loading only the relevant modules for each role and only by request.

```
// Routes
const appRoutes: Routes = [
  {path: '', redirectTo: '/home/catalogue', pathMatch: 'full'},
  {path: 'auth', loadChildren: () => import('./components/auth/auth.module').then(m => m.AuthModule)},
  {path: 'home', loadChildren: () => import('./components/home/home.module').then(m => m.HomeModule)},
  {path: 'administration', loadChildren: () =>
      import('./components/administration/administration.module').then(m => m.AdministrationModule),
    canActivate: [AuthGuard],
    data: { roles: [Role.Administrator]}},
  {path: '**', component: ErrorPageComponent}
];

@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes)
    ],
    exports: [
        RouterModule
    ],
  providers: []
})
```

Figure 29 App Router Module

Figure 29 depicts the implementation of *lazy-loading* within the application's *Router Module*. The modules are imported once the specific route is accessed. In order to restrict the unauthorized access to the administration route, the *canActivate* parameter is attached to the service *AuthGuard* which implements the interface *CanActivate*.

The *wildcard* '**' route is describing, that any path from the base URL call that is not matching with any of the settled paths will determine the rendering of the *ErrorPage* component in the router-outlet of the root – *AppComponent*, *(Figure 30).*

Figure 30 Html for Root Component

Figure 31 represents a solution for the catalogue page using resolvers to improve the user experience. The *ProductsResolver* and *DonationProgramsCatalogueResolver* are injected in the resolve object with a specific key. So, the data will be fetched from the API under that key and parsed into the route before the component is rendered.
The *DonationProgramsCatalogueResolver* implementation can be seen in *Figure 32.*

```
const homeRoutes: Routes = [
  {path: '',
    resolve: { user: UserDetailsResolver},
    component: HomeComponent, children: [
      {path: '', redirectTo: 'catalogue', pathMatch: 'full' },
      {path: 'catalogue',
        resolve: {
          products: ProductsResolver,
          donationPrograms: DonationsProgramsCatalogueResolver},
        component: CatalogueComponent },
      {path: 'place-order',
        component: PlaceOrderComponent,
        canActivate: [AuthGuard]},
      {path: 'profile', loadChildren: () =>
          import('./profile/profile.module').then(m => m.ProfileModule),
        canActivate: [AuthGuard]}
    ]}
];
```

Figure 31 homeRoutes

```
@Injectable()
export class DonationsProgramsCatalogueResolver implements Resolve<any>{
  constructor(private resourcesService: ResourcesService) {}
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<any> | Promise<any> | any {
    return this.resourcesService.getDonationProgramsCatalogue();
  }
}
```

Figure 32 DonationsProgramsCatalogueResolver

The *CatalogueComponent* has a dependency of the *ActivatedRoute*. In the constructor of the class, the *subscribe method* is called for the returned observable from the *activatedRoute* and the local variables are parsed to the response – *Figure 33*. Besides this, the figure shows an example of how the *totalElements* variable is parsed within the response's header for *products* in order to paginate and organize the elements on the *Catalogue page.* More details regarding backend pagination implementation will be offered in the upcoming sections.

```
export class CatalogueComponent implements OnInit {
products;
donationPrograms;
  constructor(private activatedRoute: ActivatedRoute) {
  this.activatedRoute.data.subscribe(
    next: response =>{
      this.donationPrograms = response.donationPrograms;
      this.products = {
        body:  response.products.body,
        totalElements: JSON.parse(response.products.headers.get('x-pagination')).totalPages};
    }
  )
  }
```

Figure 33 Catalogue Component constructor

5.1.2 Use Cases realization with UI implementation

This section will present more implementation details for the components and more focus will be placed on the code that provides the actual functionality rather than styling. For consistency reasons, the UI implementation will focus solely on the four Use Cases selected initially in order to prove their realization. Since Angular has an integrated method of simulating the *two-way data binding* inside its components through *@Output* decorated directives and events, the following sections will also present how this technique was used for the Rotaract E-shop application. Nonetheless, other relevant elements that will be discussed are the *modals*, *image loader*, the *carousel* component and the *navigation* components such as *sidebars*. As discussed before, some of these elements were inspired by the initial mockups – artifacts of the *Design Model.*

5.1.2.1 Register a new User

The *Angular specific element* for this particular use case is represented by the *RegisterComponent* which allows the user to create an account within the Rotaract E-shop application. Its interface implementation is based on the usage of *reactive forms and buttons* as it can be seen in *Figure 34*.

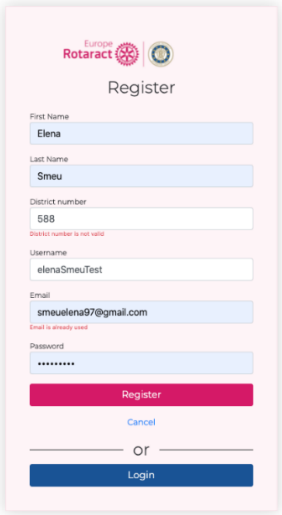


Figure 34 Register a new User UI

The *createUserForm* is built using *Angular's FormBuilder service* which allows to add validation to specific fields through the usage of *form controls*. As it can be seen in *Figure 35*, *lines 28-47*, the form is initialized after the component is set. Various validators are used within the initialization such as: required and minimum length for characters and for the email pattern.

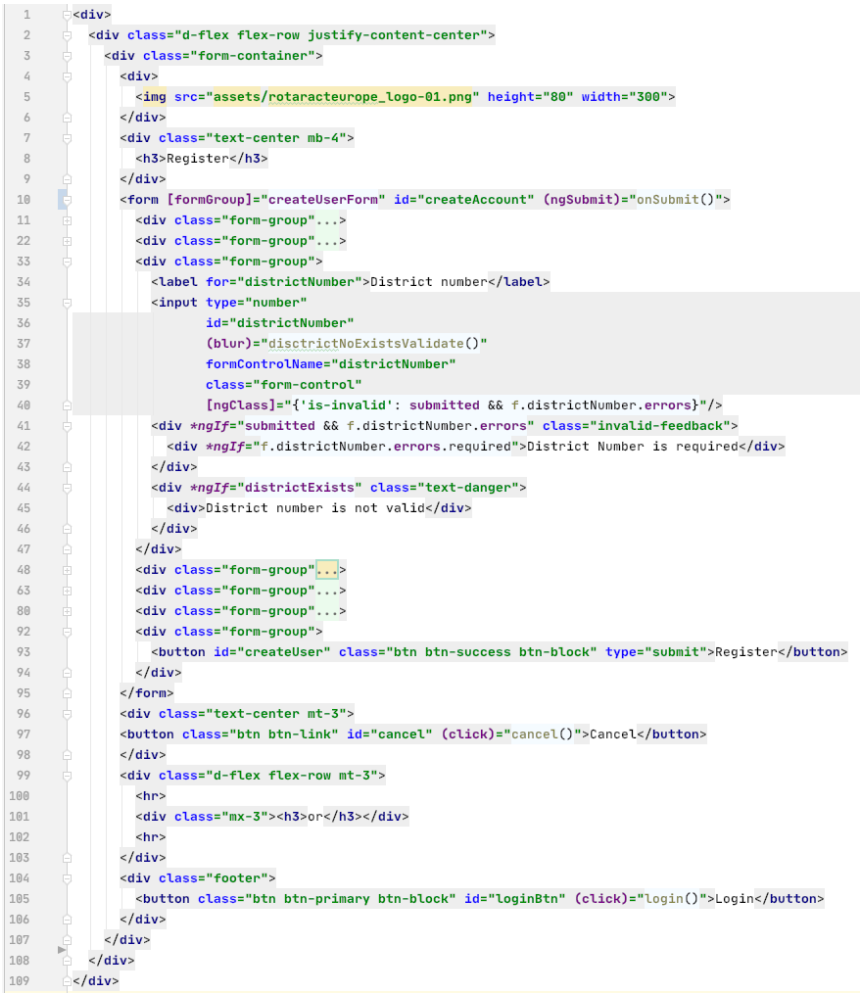
```
@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  createUserForm: FormGroup;
  submitted = false;
  usernameNotUnique = false;
  emailNotUnique = false;
  districtExists = false;
  constructor(private formBuilder: FormBuilder,
              private authService: AuthService,
              private route: Router,
              private validatorsService: ValidatorsService) {


  }
  ngOnInit(): void {
    this.createUserForm = this.formBuilder.group( controlsConfig: {
      firstName: new FormControl( formState: null,
          validatorOrOpts: {validators: [Validators.required,
              Validators.minLength( minLength: 4)]}),
      lastName: new FormControl( formState: null,
          validatorOrOpts: {validators: [Validators.required,
              Validators.minLength( minLength: 4)]}),
      districtNumber: new FormControl( formState: null,
          validatorOrOpts: {validators: [Validators.required]}),
      username: new FormControl( formState: null,
          validatorOrOpts: {validators: [Validators.required,
              Validators.minLength( minLength: 4)]}),
      email: new FormControl( formState: null,
          validatorOrOpts: {validators: [Validators.required,
              Validators.email,
              Validators.pattern( pattern: '^[a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4}$')]}),
      password: new FormControl( formState: null,
          validatorOrOpts: {validators: [Validators.required,
              Validators.minLength( minLength: 8)]})
    });
  }
```

Figure 35 Register.ts

However, during implementation of *async validators,* a bottleneck was encountered since the *ng-if directive* was not able to hold an async value within and therefore, the validation could not occur as the user was switching between the fields of the registration form. Consequently, it was decided to integrate this user experience for the fields that required async validation *(district number, email and username)* by using the *Validator endpoint* from the server side and subscribing to the response received. In *Figure 36* it can be seen how these validator methods are constructed with the purpose of changing the state of the Boolean variables that indicate the state of the validation which are used in the view and the form submission control. The methods are called when *'blur' event* occurs on the specific input field *(example: Figure 37 line 37).*

```
disctrictNoExistsValidate() {
  if(this.f.districtNumber.value){
    this.validatorsService.validateDistrictNumber(this.f.districtNumber.value).subscribe(
      next: response => { if(response) {this.districtExists = false; } else { this.districtExists = true; } }
    )
  }
}
uniquenessUsername() {
  if(this.f.username.value) {
    this.validatorsService.uniquenessUsername(this.f.username.value).subscribe(
      next: response => { if(response) { this.usernameNotUnique = true; } else {
        this.usernameNotUnique = false; } }
    )
  }
}
uniquenessEmail() {
  if(this.f.email.value) {
    this.validatorsService.uniquenessEmail(this.f.email.value).subscribe(
      next: response => {if(response) { this.emailNotUnique = true;} else {
        this.emailNotUnique = false; } }
    )
  }
}
```

Figure 36 Validators methods



```
<div>
  <div class="d-flex flex-row justify-content-center">
    <div class="form-container">
      <div>
        <img src="assets/rotaracteurope_logo-01.png" height="80" width="300">
      </div>
      <div class="text-center mb-4">
        <h3>Register</h3>
      </div>
      <form [formGroup]="createUserForm" id="createAccount" (ngSubmit)="onSubmit()">
        <div class="form-group"...>
        <div class="form-group"...>
        <div class="form-group">
          <label for="districtNumber">District number</label>
          <input type="number"
                 id="districtNumber"
                 (blur)="disctrictNoExistsValidate()"
                 formControlName="districtNumber"
                 class="form-control"
                 [ngClass]="{'is-invalid': submitted && f.districtNumber.errors}"/>
          <div *ngIf="submitted && f.districtNumber.errors" class="invalid-feedback">
            <div *ngIf="f.districtNumber.errors.required">District Number is required</div>
          </div>
          <div *ngIf="districtExists" class="text-danger">
            <div>District number is not valid</div>
          </div>
        </div>
        <div class="form-group"...>
        <div class="form-group"...>
        <div class="form-group"...>
        <div class="form-group">
          <button id="createUser" class="btn btn-success btn-block" type="submit">Register</button>
        </div>
      </form>
      <div class="text-center mt-3">
      <button class="btn btn-link" id="cancel" (click)="cancel()">Cancel</button>
      </div>
      <div class="d-flex flex-row mt-3">
        <hr>
        <div class="mx-3"><h3>or</h3></div>
        <hr>
      </div>
      <div class="footer">
        <button class="btn btn-primary btn-block" id="loginBtn" (click)="login()">Login</button>
      </div>
    </div>
  </div>
</div>
```

Figure 37 Register.html

Figure 37 shows how the *RegisterComponent* html file is using the form variable values instantiated in the *.ts file*. In *line 93* it can be seen how the *Register button* is hooked to the form which carries the *'submit' type*.

In *Figure 38* it can be seen that if the form is not valid the action ends, otherwise the constant *formUser* is declared and set to contain the current values of the form.

```
onSubmit() {
  /*
   stop if form is invalid
   */
  this.submitted = true;
  if (this.createUserForm.invalid ||
    this.usernameNotUnique ||
    this.emailNotUnique ||
    this.districtExists) {
    return;
  }
  const formUser = this.createUserForm.getRawValue();
  const userCreateRequest: User = {
    FirstName: formUser.firstName,
    LastName: formUser.lastName,
    DistrictNumber: formUser.districtNumber,
    Username: formUser.username,
    Email: formUser.email,
    Password: formUser.password,
    Role: Role.Customer
  };
  this.authService.register(userCreateRequest);
}
```

Figure 38 OnSubmit RegisterForm

Afterwards a new user object is created and parsed as a parameter in the register method of the *AuthService* which will handle the call to the server through the *ApiAuthService (Figure 39).*

```
register(user: User) {
  this.apiAuthService.createUser(user).subscribe(
    next: (response :string )  => {
    this.toast.success(response.toString());
    this.route.navigate( commands: ['auth/login']);
},
    error: (err) => {this.toast.error(err.error.message);});
}
```

Figure 39 AuthService register

5.1.2.2 Cart

The idea of implementing the cart feature only in the *Presentation Tier* was related to the fact that this entity lives just during the sessions. Thus, the system does not need to keep track or store the cart for every individual user for a long period of time since after the session expired the items might not be needed anymore or might be outdated.

The *CartComponent* is an exported component of the *sharedModule* and it is used as a *widget* inside the application *(Figure 40).* As it can be seen below, the *CartService* class is acting as a middleware between the *CartComponent* and the local storage of the browser.

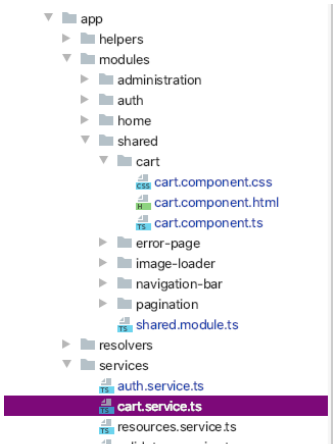


Figure 40 Cart File Organization

The *CartComponent* was designed using the simple bootstrap modal layout. The products added in the cart are displayed as containers which provide individual access to functionality for each – *Figure 41*

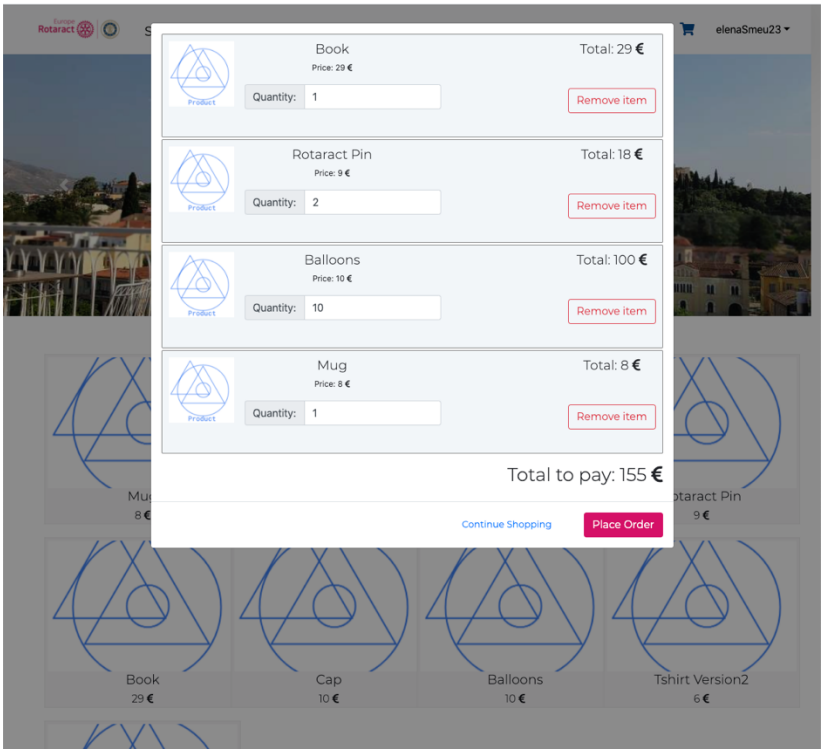


Figure 41 CartComponent UI

The *CartService* main functions are adding product items, removing product items, changing the quantity of already added products and updating the cart and the total price after each change – *Figure 42*.

The same figure shows how the service is using the *isAuthenticated get* method of the *AuthService* to simulate a *guard* for the object if the user is not authenticated when adding a product to the cart which thereafter will redirect the user to the *login page component*. However, the items are kept in the local storage of the session and retrieved by the cart after the user authenticates.

```
@Injectable({
  providedIn: 'root'
})
export class CartService {

  constructor(private toast: ToastrService,
              private authService: AuthService,
              private route: Router) {
  }

  addProductItem(item: ProductItem) {
    const cartList = this.cart();
    if( cartList.products.find( predicate: product => product.item.ProductId === item.item.ProductId)) {
      for ( let product of cartList.products){
        if ( product.item.ProductId === item.item.ProductId ) {
          product.quantity = product.quantity + item.quantity;
        }
      }
    } else {
      cartList.products.push(item);
    }
    this.updateCart(cartList);

    if (this.authService.isAuthenticated) {
      this.toast.success( message: item.quantity + ' ' +
        item.item.Name +
        ' has been added in the cart');
    } else {
      this.route.navigateByUrl( url: 'auth/login');
    }
  }
  removeProductItem(item: ProductItem) {...}
  changeQuantity(productItem: ProductItem, quantity: number) {...}
  updateCart(cartList) {
    /*
    set the storage item cart to the new Json string value
     */
    localStorage.setItem('cart', JSON.stringify(cartList));
    if ( this.authService.isAuthenticated) {
      this.toast.success( message: 'The cart has been updated, new total: ' +
        this.getTotal());
    }
  }
  emptyCart() {...}
  getTotal() {...}
  cart() {...}
}
```

Figure 42 Cart Service

5.1.2.3 Place an order

Placing an order functionality is a focal point of this application and therefore, the realization of this use case was mandatory to be described also in the implementation of the presentation tier. Linked to the cart implementation described above, when the *place order button* is pressed, the user is redirected to the *place-order page component* using the router service.

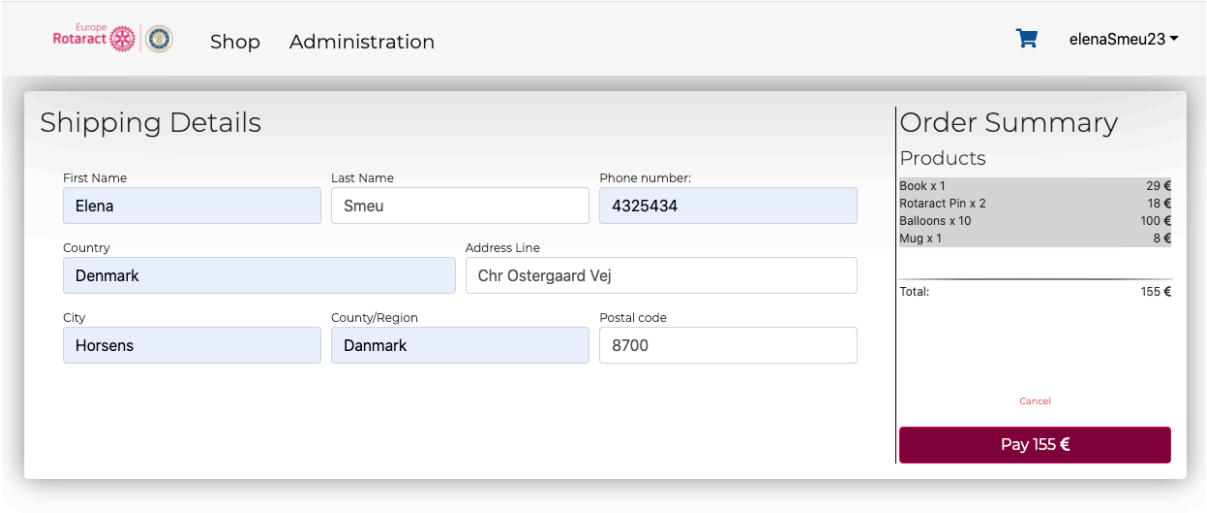


Figure 43 Place an order UI

In *Figure 43* the *PlaceOrderComponent* is designed using the same technique based on reactive form with validation and buttons except that in this one there are no async validators required.

After the fields from *Figure 43* are filled by the user and the button *Pay* is pressed, the order is constructed by *mapping* the current *product items* from the cart to *order items* using the *productIds* and the *quantities (Figure 44, lines 42-47)* and the total price for the order will be calculated further by the server-side logic, retrieved as a result of the calls and thereafter displayed on the client-side interface.

```
36  onSubmit() {
37    this.submitted = true;
38    if(this.shippingDetailsForm.invalid) {
39      return;
40    }
41    const obj = this.shippingDetailsForm.getRawValue();
42    const orderItems = this.itemList().products.map( product => {
43      return {
44        product: {ProductId: product.item.ProductId},
45        Quantity: product.quantity };
46    });
47    const order: Order = {OrderItems: orderItems...}
61    this.resourcesService.placeOrder(order).subscribe(
62      next: response => { this.cartService.emptyCart();},
63      error: (err) => {
64        this.submitted = false;
65        },
66      complete: () => this.router.navigate( commands: ['home/profile/orders'])
67    )
68  }
```

Figure 44 OnSubmit of Place Order

After the order is successfully sent and saved, the cart is emptied and the user is redirected to the profile's orders *(Figure 44 line 66 - Figure 45)*.

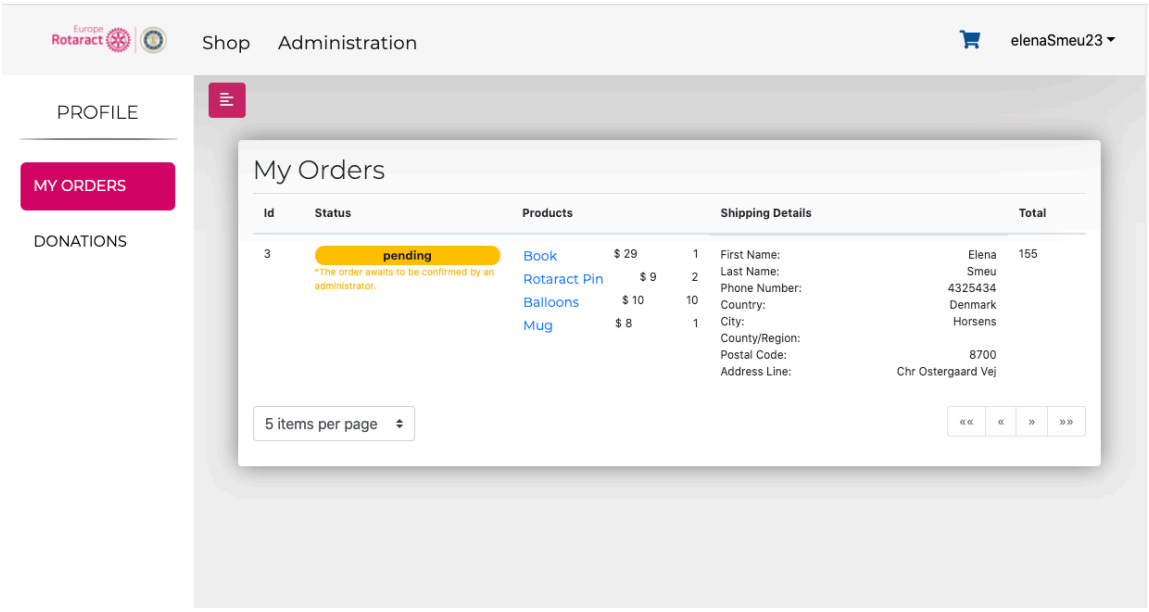


Figure 45 Profile's Orders

5.1.2.4 Manage Orders

As more customers are placing order in the system, the administrator must be able to manage them according to the system's use case and this section presents how this is achieved through the UI implementation. The management of orders functionality is placed in the administration module and it includes at first an overview of all orders stored in the system which can be navigated by tabs that represent the status of the order and also by pagination if the number of orders requires it *(Figure 46)*.

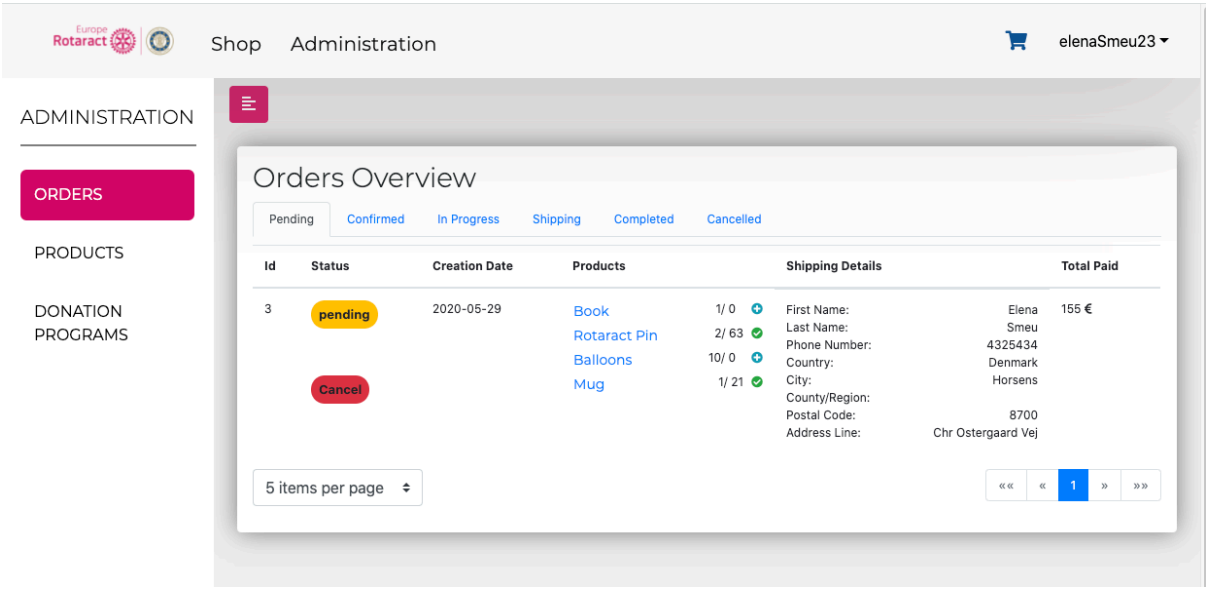


Figure 46 Orders Overview

The implementation within the *OrdersOverviewComponent* is shown in *Figure 47* where a list is constructed as a *ng-boostrap navigation component*. Each element from the list represents a tab and a content with the name on the tab being set through the directive *ngbNavLink* and the content using *ng-template* with the directive *ngbNavContent* to insert the *OrdersComponent* and parse the specific status.

```
<div class="table-container mt-3">
  <h3>Orders Overview</h3>
  <ul ngbNav #nav = "ngbNav"
      [(activeId)]="active"
      [destroyOnHide]="false"
      class="nav-tabs">
    <li [ngbNavItem]="1" [destroyOnHide]="true">
      <a ngbNavLink>Pending</a>
      <ng-template ngbNavContent>
        <app-orders [status]="status.PENDING"></app-orders>
      </ng-template>
    </li>
    <li [ngbNavItem]="2" [destroyOnHide]="true">
      <a ngbNavLink>Confirmed</a>
      <ng-template ngbNavContent>
        <app-orders [status]="status.CONFIRMED"></app-orders>
      </ng-template>
    </li>
    <li [ngbNavItem]="3" [destroyOnHide]="true">
      <a ngbNavLink>In Progress</a>
      <ng-template ngbNavContent>
        <app-orders [status]="status.INPROGRESS"></app-orders>
      </ng-template>
    </li>
    <li [ngbNavItem]="4" [destroyOnHide]="true">
      <a ngbNavLink>Shipping</a>
      <ng-template ngbNavContent>
        <app-orders [status]="status.SHIPPING"></app-orders>
      </ng-template>
    </li>
    <li [ngbNavItem]="5" [destroyOnHide]="true">
      <a ngbNavLink>Completed</a>
      <ng-template ngbNavContent>
        <app-orders [status]="status.COMPLETED"></app-orders>
      </ng-template>
    </li>
    <li [ngbNavItem]="6" [destroyOnHide]="true">
      <a ngbNavLink>Cancelled</a>
      <ng-template ngbNavContent>
        <app-orders [status]="status.CANCELLED"></app-orders>
      </ng-template>
    </li>
  </ul>
  <div [ngbNavOutlet]="nav" class="mt-2"></div>
</div>
```

Figure 47 OrdersOverviewComponent.html

As mentioned above and described by its name, the component from *Figure 47* offers just an overview of all orders. The functionality for managing each order is hold by the *OrdersComponent* which is constructed as a table of which content changes based on pagination and fetched data. The component declares an *@Input* decorated variable status of which value is set by its parent – *OrdersOverviewComponent* as it can be visualized in *Figure 48, line 13*.

The *OrdersComponent* initialization shown in *Figure 48* is described by fetching the orders from the server based on their status and pagination using the same method of subscribing to the response.

```
@Component({
  selector: 'app-orders',
  templateUrl: './orders.component.html',
  styleUrls: ['./orders.component.css']
})
export class OrdersComponent implements OnInit {
  @Input() status;
  page = 1;
  pageSize = 5;
  totalElements;
  orders;
  modalOptions;
  constructor(private resourcesService: ResourcesService,
              private modalService: NgbModal,
              private toastrService: ToastrService) {
    this.modalOptions = {
      centered: true,
      size: 'lg'
    }
  }

  ngOnInit(): void {
    this.getOrders();
  }
  getOrders() {
    this.resourcesService.getOrders(this.status, this.page, this.pageSize).subscribe(
      next: response => {
        this.orders = response.body;
        this.totalElements = JSON.parse(response.headers.get('x-pagination')).totalPages;}
    )
  }
```

Figure 48 OrdersComponent.ts -Constructor and Initialization

Additionally, in *line 35*, the *totalElements* variable is set based on the *Header 'x-pagination' totalPages property* which is used to structure the order elements on the UI page.

For each order from the table, the status is shown as a *badge bootstrap element* which also holds the functionality to move to the next status. *Figure 49* illustrates the usage of the *ngClass directive* to show the badge type depending on the *OrderStatus*.

```
16    <a (click)="changeStatus(order)"
17      class="badge p-2 badge-pill"
18      id="changeStatus{{order.OrderId}}"
19      [ngClass]="{'badge-warning': order.Status==='PENDING',
20     'badge-success': order.Status ==='CONFIRMED'|| order.Status === 'COMPLETED',
21     'badge-info': order.Status === 'INPROGRESS',
22     'badge-light': order.Status === 'SHIPPING',
23     'badge-secondary': order.Status === 'CANCELLED'}">
24     {{order.Status | lowercase}}</a>
```

Figure 49 Status Badge in OrdersOverview.html

When the badged link is pressed, the *changeStatus* method is called for the specific order which can be noticed in *Figure 50*. Implementing the method using a switch case allows to choose the right approach for each specific status. Thus, the *completed* or *cancelled* orders cannot have their status changed since there is no further state – therefore, a toaster warning will appear if the administrator tries to change the status for these orders as it can be seen below.

```
38  changeStatus(order) {
39    switch (order.Status) {
40      case Status.PENDING :  this.toastrService.warning( message: 'Please check the products availability'); break;
41      case Status.CONFIRMED : this.resourcesService.changeOrderStatus( order: { OrderId: order.OrderId, Status: Status.INPROGRESS}).subscribe(
42          next: res => { this.getOrders(); this.toastrService.success( message: 'Order has been moved to In Progress'); }),
43        (err) => { this.toastrService.error( message: 'Something went wrong.')}; break;
44      case Status.INPROGRESS : this.resourcesService.changeOrderStatus( order: { OrderId: order.OrderId, Status: Status.SHIPPING}).subscribe(
45          next: res => { this.getOrders(); this.toastrService.success( message: 'Order has been moved to Shipping'); }),
46        (err) => { this.toastrService.error( message: 'Something went wrong.')}; break;
47      case Status.SHIPPING : this.resourcesService.changeOrderStatus( order: { OrderId: order.OrderId, Status: Status.COMPLETED}).subscribe(
48          next: res => { this.getOrders(); this.toastrService.success( message: 'Order has been moved to Shipping'); }),
49        (err) => { this.toastrService.error( message: 'Something went wrong.')}; break;
50      default: this.toastrService.warning( message: 'Not allowed!')
51    }
52  }
53  cancelOrder(order) {
54    this.resourcesService.changeOrderStatus( order: { OrderId: order.OrderId, Status: Status.CANCELLED}).subscribe(
55        next: res => { this.getOrders(); this.toastrService.success(res.toString()); },
56        error: (err) => { this.toastrService.error( message: 'Something went wrong.')}
57      );
58  }
```

Figure 50 ChangeStatus of an order in OrdersComponent.ts

5.1.2.5 Manage Donation Programs

The idea of including donation programs in the solution is one of the elements that makes this application stand out when compared to other e-commerce platforms and it was also suitable for the profile of the Rotaract Europe organization. This section describes how the UI implementation supports the realization of the *Manage Donation Programs use case.*

The presentation tier element defining this case was designed using a single modal component for both adding and editing a program, and also a table with pagination for navigation through the donation programs inside the *DonationProgramsOverview (Figure 51).*

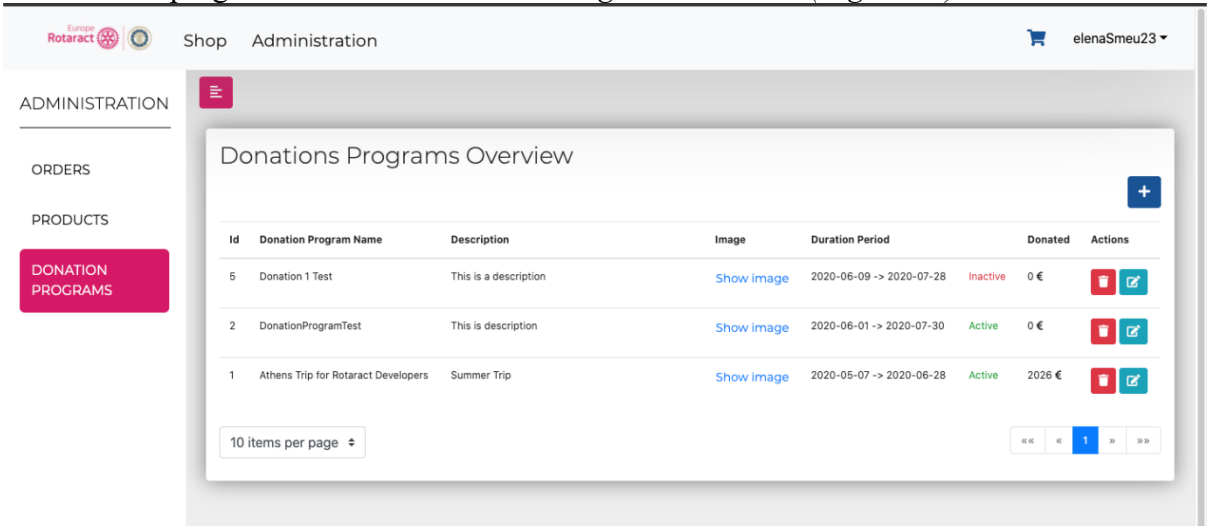


Figure 51 DonationProgramsOverview UI

The *modals* are made as stand-alone components that are declared inside an *ng-template*. An example of such construction can be seen in *Figure 52 (lines 52-55 and lines 66-78)*. Additionally, in *line 4* of the same figure it can be noticed how clicking the *add button* triggers the call of the *open method* which is taking as argument a *target element* declared with the *ng-template*.

```
1  <div class="table-container mt-3">
2    <h3>Donations Programs Overview</h3>
3    <div class="float-right mb-3">
4      <button class="btn btn-primary" id="add" (click)="open(addDonationProgram)">
5        <i class="fas fa-plus"></i>
6      </button>
7    </div>
8    <div class="table-responsive mb-2">
9      <table class="table">
10       <thead...>
20       <tbody>
21       <tr *ngFor="let donationProgram of donationPrograms">
22         <td>{{donationProgram.DonationProgramId}}</td>
23         <td>{{donationProgram.DonationProgramName}}</td>
24         <td><div class="overflowed overflowed-ellipsis-word">
25           <p>{{donationProgram.Description}}</p></div></td>
26         <td...>
32         <td>{{date(donationProgram.StartDate)}} -> {{date(donationProgram.EndDate)}}</td>
33         <td...>
37         <td>{{donationProgram.Total}} <i class="fas fa-euro-sign"></i> </td>
38         <td>
39           <button class="btn btn-sm btn-danger mr-1" id="delete{{donationProgram.DonationProgramId}}"
40                   (click)="delete(donationProgram.DonationProgramId)">
41           <i class="fas fa-trash"></i>
42           </button>
43           <button class="btn btn-sm btn-info" id="edit{{donationProgram.DonationProgramId}}"
44                   (click)="open(editDonationProgram)">
45             <i class="fas fa-edit"></i>
46           </button>
47         </td>
48         <ng-template...>
51         <ng-template #editDonationProgram>
52           <app-donationProgram-form [donationProgram]="donationProgram"
53                                     (donationProgramChange)="getDonationPrograms()">
54           </app-donationProgram-form>
55         </ng-template>
56       </tr>
57       </tbody>
58     </table>
59   </div>
60   <app-pagination...>
65 </div>
66 <ng-template #addDonationProgram>
67   <app-donationProgram-form [donationProgram]="null"
68                             (donationProgramChange)="getDonationPrograms()">
69   </app-donationProgram-form>
70 </ng-template>
```

Figure 52 DonationProgramsOverviewComponent.html

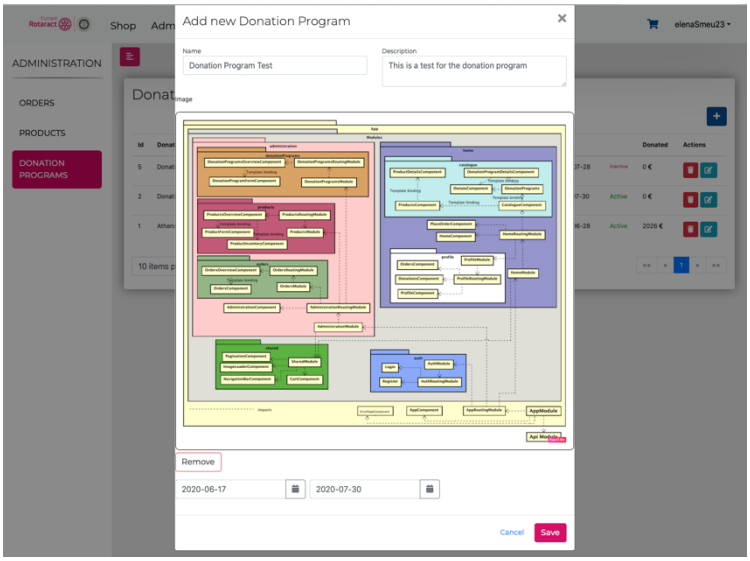


Figure 53 AddDonationProgram Modal UI

The *DonationProgramFormComponent* from *Figure 53* is constructed using a reactive form, an image loader, a datepicker that describes the duration period of the program and buttons for cancel and save.

The *DonationProgramFormComponent* relies on the *@Input* decorated variable *donationProgram* to identify its type and the object it refers to from the parent component *DonationProgramsOverviewComponent*. If the *@Input* is null, then this triggers the component to represent an adding form for that resource, otherwise the form will become an update one filled with the specific properties of the provided resource which in turn, will become subject for change. This is represented in the *Figure 54*, *lines 35 to 54* which define one of the component's *life cycle*: *OnInit* that was extensively used during development. Though this method, during the initialization of the component, different properties are set for the two purposes previously defined.

```
14  export class DonationProgramFormComponent implements OnInit {
15  @Input() donationProgram: DonationProgram;
16  @Output() donationProgramChange = new EventEmitter<any>();
17  donationProgramForm: FormGroup;
18  image: string;
19  submitted = false;
20  header;
21    hoveredDate: NgbDate | null = null;
22    fromDate: NgbDate | null;
23    toDate: NgbDate | null;
24    constructor(private resourcesService: ResourcesService,
25                private formBuilder: FormBuilder,
26                private modalService: NgbModal,
27                private dateFormat: DateFormat,
28                private calendar: NgbCalendar,
29                public formatter: NgbDateParserFormatter,
30                private toastrService: ToastrService) {...}
33
34    ngOnInit(): void {
35      if (this.donationProgram) {
36        this.header = 'Update programme: ' + this.donationProgram.DonationProgramName;
37        this.donationProgramForm = this.formBuilder.group( controlsConfig: {
38          donationProgramName: new FormControl(this.donationProgram.DonationProgramName,
39            validatorOrOpts: {validators: Validators.required}),
40          description: new FormControl(this.donationProgram.Description)
41        });
42        this.image = this.donationProgram.ImageRef;
43        this.fromDate = this.dateFormat.dateFromString(this.donationProgram.StartDate);
44        this.toDate = this.dateFormat.dateFromString(this.donationProgram.EndDate);
45      } else {
46        this.header = 'Add new Donation Program';
47        this.donationProgramForm = this.formBuilder.group( controlsConfig: {
48          donationProgramName: new FormControl( formState: null,
49            validatorOrOpts: {validators: Validators.required}),
50          description: new FormControl( formState: null)
51        });
52        this.fromDate = this.calendar.getToday();
53        this.toDate = this.calendar.getNext(this.calendar.getToday(), period: 'd', number: 40);
54        this.image = null;
55      }
56    }
```

Figure 54 DonationProgramFormComponent.ts

```
71   if (this.donationProgram){...} else {
89     this.resourcesService.addDonationProgram(donationProgram).subscribe(
90       next: (res :Object ) => {
91         this.donationProgramChange.emit(res);
92         this.modalService.dismissAll();
93         this.toastrService.success( message: 'A donation program has been added');
94       },
95       error: (err) => { this.toastrService.error(err.error.message); }
96     );
97   }
```

Figure 55 DonationProgramForm.ts

After the changes are made in the UI and the *submit button* of the modal form is pressed, the *onSubmit* method is called which performs the call to the backend and receives a response. As it can be seen in line 91 of *Figure 55,* the form child component will *emit an event* back to its parent component *DonationProgramsOverview*. Consequently, this defines how *two-way data binding* is performed and applied inside the presentation tier of the application. Additionally, in *Figure 55 line 92* it can also be noticed how the modal is dismissed after their usage.

5.1.3 API Module

This section will describe the implementation of the *ApiModule* in the Angular application. *Figure 56* shows how various files and packages are organized inside the module. The services that host the *RESTful API methods* are placed in directories of which naming indicates the subsequent functionality – such as *'auth'* for *authentication* and *authorization*.

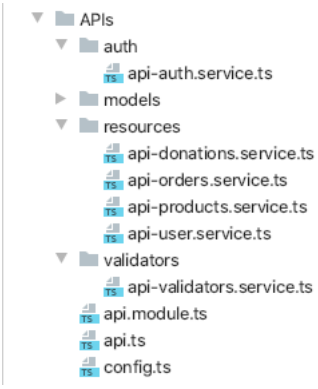


Figure 56 File organization in ApiModule

Figure 57 describes the implementation of the *ApiModule* class. In *lines 23-28* the method *forRoot* returns a *ModuleWithProviders* and takes as parameter a *configurationFactory* method which returns a *Config object* used by the *api-services* within the module. The factory method is used by the *providers property* of the *ApiModule* to setup the Config using the *useFactory* parameter.

```
11  @NgModule({
12    imports: [],
13    declarations: [],
14    exports: [],
15    providers: [...]
21  })
22  export class ApiModule {
23    public static forRoot(configurationFactory: () => Config): ModuleWithProviders {
24      return {
25        ngModule: ApiModule,
26        providers: [ { provide: Config, useFactory: configurationFactory }]
27      };
28    }
29    constructor(@Optional() @SkipSelf() parentModule: ApiModule,
30                @Optional() httpClient: HttpClient) {...}
38  }
39
```

Figure 57 ApiModule Implementation

In *Figure 58 – lines 43 to 48* – the implementation of the *apiConfigFactory* is described. Inside the function, the constant variable *params* defines another variable *ConfigParameters* constructed with the *basePath* property of the apiUrl – which is set from the environment and the *accesToken* property – which is set with the *user's token* found in the session storage.

The method returns a new *Config* object that takes the *params* variable as parameter.
Afterwards, it can be seen in *line 26* how the factory is taken as parameter in the *forRoot method* when the *ApiModule* is imported.

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
      NgbModule,
      BrowserModule,
      HttpClientModule,
      AppRouterModule,
      ApiModule.forRoot(apiConfigFactory),
      BrowserAnimationsModule,
      ToastrModule.forRoot({positionClass: 'toast-bottom-left'...}),
      SharedComponentsModule
  ],
  providers: [
    AuthService,
    ApiValidatorsService,
    { provide: HTTP_INTERCEPTORS, useClass: HttpErrorInterceptor, multi: true}
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
export function apiConfigFactory() {
  const params: ConfigParameters = {
    basePath: environment.apiUrl,
    accessToken: sessionStorage.getItem( key: 'key')
  };
  return new Config(params);
}
```

Figure 58 Import of ApiModule in AppModule

Figure 59 shows how the *ApiOrdersService* is implemented. In *lines 9 to 12*, it can be seen that the service declares and instantiates a *Config object* which in the constructor has an *@Optional dependency*. The object actually refers to the one returned by the *AppModule* and if it exists the config will be used by the *apiService*.
In *line 10* can be seen the dependency of *HttpClient* service provided by Angular which is using RESTful methods such as: *GET*, *POST*, *PUT*, *DELETE* to make requests to the server. For each endpoint where its required authorization, a header is made and parsed in the request's headers, example in *Figure 59*.

```
@Injectable({
  providedIn: 'root'
})
export class ApiOrdersService {
  public configuration = new Config();
  constructor(private httpClient: HttpClient, @Optional() config: Config) {
    if (config) {this.configuration = config; }
  }
  getOrders(status, page, pageSize) {
    const headers = {
      Authorization: `Bearer ${this.configuration.accessToken}`
    };
    return this.httpClient.get( url: `${this.configuration.basePath}api/order/${status}?pageResourceParameters.pageNumber=
      options: {headers, observe: "response"});
  }
  placeOrder(order) {
    const headers = {
      Authorization: `Bearer ${this.configuration.accessToken}`
    };
    return this.httpClient.post( url: `${this.configuration.basePath}api/order`, order, options: {headers});
  }
  changeOrderStatus(order) {
    const headers = {
      Authorization: `Bearer ${this.configuration.accessToken}`
    };
    return this.httpClient.put( url: `${this.configuration.basePath}api/order`, order, options: {headers});
  }
}
```

Figure 59 ApiOrdersService implementation

5.2 Business logic Tier

This section of the *Implementation Model* describes various objects of implementation relevant not only for the chosen Use Cases but for the application per se on various matters such as: *endpoints, security, data access, pagination* or interesting solutions to development problems. Apart from the technology choice for this tier's implementation which was described previously, it is relevant to mention that some snippets supporting the argumentation represent code written using C# or LINQ queries for data manipulation.

5.2.1 API Endpoints

An essential element for the communication between the presentation tier and the business logic tier is represented by the RESTful API endpoint which – on the server side – responds to UI requests through an *HttpResponseMessage* after the request is mapped to a suitable method implemented by the *Controllers*. As described previously, the client application ultimately uses these endpoints to access and alter the data stored in the database.

```
[AllowAnonymous]
[System.Web.Http.HttpGet]
[System.Web.Http.Route(template: "donationProgram/catalogue")]
[CacheOutput(ClientTimeSpan = 30, ServerTimeSpan = 30)]
1 reference
public HttpResponseMessage GetActiveDonations()
{
    try
    {
        var donationProgramsFromRepo = _rotaractRepository.GetActiveDonationPrograms();

        return Request.CreateResponse(HttpStatusCode.OK, donationProgramsFromRepo);
    }
    catch (Exception e)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, e);
    }
}
```

Figure 60 GET DonationProgram catalogue endpoint

Figure 60 depicts one of such back-end endpoint methods *(REST GET)* that allows the user to retrieve the active donation programs from the database. The return type message includes the status code for the *CreateResponse* method as well as a message in the form of a *string* or a *JSON formulated object*. All endpoints need to be decorated with *tags* such as *[AllowAnonymous]* which allows anonymous requests with no authentication token included, tags that define the type of the call and *routing tags* to specify an alternate address other than the prefix located on top of the *Controller* class. Additionally, seen in the figure is the *caching setting tag* which predefines a timeframe for how the endpoint is accessed.

Another type of endpoint is defined in *Figure 61* – for deletion purposes. In this example, the endpoint is decorated with the *HttpDelete tag* but also with the routing tag which in this case includes the variable that relate to a specific id for a *DonationProgram* to be deleted. As it can be seen in the authorization tag, the predefined user role that has access to this endpoint's operation is the *Administrator*.

```
[Authorize(Roles = "Administrator")]
[System.Web.Http.HttpDelete]
[System.Web.Http.Route(template: "donationProgram/{donationProgramId}")]
1 reference
public HttpResponseMessage DeletedDonation(int donationProgramId)
{
    try
    {
        if (_rotaractRepository.DeleteDonationProgram(donationProgramId))
        {
            _rotaractRepository.Save();
            return Request.CreateResponse(HttpStatusCode.OK, value: "Donation deleted from the system");
        }

        return Request.CreateResponse(HttpStatusCode.NotFound, value: "Donation Program Doesn't Exist");
    }
    catch (Exception e)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, e);
    }
}
```

Figure 61 DELETE Donation Program endpoint

5.2.2 Authorization

To provide a secured way to access for all endpoints, secure tokens with a lifetime of 24 hours are used and generated on the server side and sent to the client side after each login. This approach uses and applies *Claims* for each user requested from the database context. Most of the endpoints using *Authorization* only need the token to grant the user with data specific to the role such as: customer's orders or the list of all orders stored in the system for the administration's overview.

As it can be seen in *Figure 62*, the class *MyAuthorizationServerProvider* which inherits from *OAuthAuthorizationServerProvider* defines an override of the *async method GrantResourceOwnerCredentials* which is using the repository context to identify a user given the *username* and *password.* In the case when there is no match with an existing row in the database, an *invalid grant* error is returned, otherwise an identity is created for the user with the information from the database which is thereafter validated using the async method: *ValidateClientAuthentication.*

```
public class MyAuthorizationServerProvider : OAuthAuthorizationServerProvider
{
    0 references
    public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
    {
        context.Validated();
    }

    0 references
    public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
    {
        using (var _repo = new RotaractServrerContext())
        {

            var account = _repo.Users.FirstOrDefault(n => n.Username == context.UserName && n.Password == context.Password);

            if (account == null)
            {
                context.SetError("invalid_grant", "Provided Username and password is incorrect");
                return;
            }

            var identity = new ClaimsIdentity(context.Options.AuthenticationType);
            identity.AddClaim(new Claim(ClaimTypes.Role, account.Role));
            identity.AddClaim(new Claim("Username", account.Username));
            identity.AddClaim(new Claim("UserId", account.UserId.ToString()));
            identity.AddClaim(new Claim("FirstName", account.FirstName));
            identity.AddClaim(new Claim("LastName", account.LastName));
            identity.AddClaim(new Claim("Email", account.Email));
            identity.AddClaim(new Claim("DistrictNo", account.DistrictNumber.ToString()));
            context.Validated(identity);
        }
    }
}
```

Figure 62 MyAuthorizationServerProvider.cs

5.2.3 Data Access

As discussed previously, between the *Controller's endpoints logic* and the *repository Db Context* a layer of abstraction is established by implementing the *Unit of work* design pattern and making each *Controller* class point to the interface of the *RotaractRepository*. As it can be seen in *Figure 63*, a *_context* attribute is declared as *private readonly* to reference the *RotaractServerContext* where data is further mapped into tables and rows of the MSSQL database.

```
private readonly RotaractServrerContext _context;
5 references
public RotaractRepository(RotaractServrerContext context)
{
    this._context = context;
}
```

Figure 63 RotaractRepository constructor

The *RotaractRepository* class therefore acts as the bridge between the server application and the database tier. It implements all of the logic required by the endpoints to process the requests from the client and to perform the necessary retrieval and manipulation of database data using either LINQ expressions or SQL queries. As it can be noticed in *Figure 64* which describes the *GetActiveDonationPrograms* method from the repository class, the method first filters a sequence of values based on a predicate and then returns these values in a list.

```
3 references
public List<DonationProgram> GetActiveDonationPrograms()
{
    DateTime today = DateTime.Now;
    List<DonationProgram> donationPrograms = _context.DonationPrograms.Where(n=> n.StartDate <= today && n.EndDate >= today).ToList();
    return donationPrograms;
}
```

Figure 64 GetActiveDonationPrograms method

5.2.4 Pagination implementation

This implementation appeared to be an appropriate solution during the development of UI components as more content *(in the form of products and donation programs)* was added to the pages. These resources required an optimal display solution that will also improve the performance and reduce the rendering time of the components. Instead of having the endpoints request a specific number of objects, the method *GetProducts* from *Figure 65,* makes use of the *pageResourceParameters* sent through the *HttpClient* request to retrieve a *PagedList* collection of a certain count of these objects alongside the size and page number *(included in the body of the response)*.

```
2 references
public PagedList<Product> GetProducts(PageResourseParameters pageResourseParameters)
{
    if (pageResourseParameters == null)
    {
        throw new ArgumentNullException(nameof(pageResourseParameters));
    }

    var collection = _context.Products.ToList();

    return PagedList<Product>.Create(collection,
        pageResourseParameters.PageNumber,
        pageResourseParameters.PageSize);
}
```

Figure 65 PageResouceParameters

The implementation of the *PagedList* class constructor and of the *Create* method called by the endpoint can be seen in *Figure 66*.

```
4 references
public PagedList(List<T> items, int count, int pageNumber, int pageSize)
{
    TotalCount = count;
    PageSize = pageSize;
    CurrentPage = pageNumber;
    TotalPages = (int)Math.Ceiling(count / (double)pageSize);
    AddRange(items);
}
6 references
public static PagedList<T> Create(List<T> source, int pageNumber, int pageSize)
{
    var count = source.Count();
    var items = source.Skip((pageNumber - 1) * pageSize).Take(pageSize).ToList();
    return new PagedList<T>(items, count, pageNumber, pageSize);
}
```

Figure 66 PagedList class

5.3 Data Tier

Briefly discussed in other sections of the report, the data tier is the segment of the system that defines how the entity objects derived from the *Domain Model* and by now mapped into objects of design and implementation are *transformed* into the data of tables and rows from a chosen persistence environment – in this case, the MSSQL database. This section will describe how this transformation was achieved using *SQL language*, will offer an extended view upon some relevant multiplicities and constraints of the *ER diagram* and will also present some interesting solutions that surfaced during the development. Nonetheless, the main focus of this section will be placed on the *Place an order* functionality (selected mostly because it includes the most interesting aspects related to the data tier but also because it defines the essence of an e-commerce solution).

5.3.1 Entity Framework with MSSQL

Using these technologies together allowed for the application to be developed using a *code-first approach* especially during the first part of the implementation of the server-side. This approach is made possible by the usage of the Entity Framework which maps the *Entities* classes into *DbSets* which refer to tables in the MSSQL database for which entries are added as rows. Therefore, this approach was also convenient on the reason that there was no predefined schema required by the stakeholders and thus it was a subject of multiple changes over the course of the implementation.

An example of an entity class is the *User entity* which is depicted in *Figure 67* with all the property columns in the form of methods decorated with tags to specify *required columns (Not Null), identifiers (PK)* and *uniqueness (IsUnique)*.

```
9 references
public class User
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    4 references
    public int UserId { get; set; }
    [Required]
    [Index(IsUnique = true)]
    4 references
    public string Username { get; set; }
    [Required]
    [Index(IsUnique = true)]
    6 references
    public string Email { get; set; }
    [Required]
    1 reference
    public string FirstName { get; set; }
    [Required]
    1 reference
    public string LastName { get; set; }
    [Required]
    2 references
    public int DistrictNumber { get; set; }
    [Required]
    2 references
    public string Password { get; set; }
    [Required]
    1 reference
    public string Role { get; set; }

}
```

Figure 67 User Entity Framework class

However, after the final version of the database schema was created, there was an increased need to have more control over how the database is maintained and thus to depend less on the

auto-generated SQL code from migrations. This need was also determined by some problems with working with the framework such as *foreign keys* not being created correctly or changed decorators not being updated in the database. Additionally, there were identified issues with regards to memory allocations which could potentially affect the performance depending on how the application would scale: Entity Framework automatically allocated 200 kilobytes for fields that would never exceed 50 characters instead of a more appropriate 5 kilobytes using an alternative implementation. (*[N]Varchar(4000) and Performance – SQLServerCentral*)

```
CREATE TABLE [dbo].[Users](
    [UserId] [int] IDENTITY(1,1) NOT NULL,
    [DistrictNumber] [int] NOT NULL,
    [Username] [varchar](50) NOT NULL,
    [Email] [varchar](50) NOT NULL,
    [FirstName] [varchar](50) NOT NULL,
    [LastName] [varchar](50) NOT NULL,
    [Password] [varchar](50) NOT NULL,
    [Role] [varchar](20) NOT NULL,
 CONSTRAINT [PK_Users] PRIMARY KEY CLUSTERED
(
    [UserId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Figure 68 SQL Users Table Creation Query

The solution to be used alongside the framework was to define a schema using the ER diagram already created as skeleton and the Microsoft SQL Management Studio as tool. This allowed for a DDL script file to be written in order to initialize all the tables required, establish PKs and FKs and define any necessary triggers. In *Figure 68*, the creation of the *Users table* can be seen along with columns that have a conveniently specified varchar length and few necessary constraints. To accommodate the necessary check for an existing district number required by the *Register as new user Use Case*, the *Users table* had to be altered in order to reference the *DistrictNoes table* with a foreign key and *Figure 69* displays how this was implemented using the SQL language.

```
ALTER TABLE [dbo].[Users]  WITH CHECK ADD  CONSTRAINT [FK_Users_has] FOREIGN KEY([DistrictNumber])
REFERENCES [dbo].[DistrictNoes] ([DistrictNumber])
GO
ALTER TABLE [dbo].[Users] CHECK CONSTRAINT [FK_Users_has]
GO
```

Figure 69 User to District number table constraint creation query

5.3.2 Concurrency Control

The main challenge during the development of this tier was to solve database consistency issues. In the case of placing an order, the question was put this way: *what would happen if two customers place orders at almost the same time including product/s that normally would have enough inventory for an individual order but not for both?* Without any form of custom-made concurrency control and using only the *Entity Framework* procedure, both customers will receive a confirmation of their orders which ultimately will result in data inconsistency of the that specific product's inventory, orders that cannot be actually fulfilled and an administrator who is not aware of neither issues so that the administrator can reliably solve them.

```
string ProductQuery = "SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;" +
                      " BEGIN TRANSACTION;" +
                      " SELECT * " +
                      " FROM   dbo.Products WITH(XLOCK); ";
string CommitQuery = " COMMIT TRANSACTION;";
```

Figure 70 Transaction Query - Products Table

PostNewOrder method of the *RotaractRepository* is the only method that required more than what the Entity Framework could provide for the communication with MSSQL database and as it can be seen in *Figure 70*, this method uses *Transaction queries* to address the concurrency issue. The implementation above uses these two queries to *begin* and *commit* the transactions: the first one selects all columns from the *Products table* and it does so using an *Xlock* at a *Serializable level* which puts any other following requests that have to deal with the *Products table* to the database in a queue until the *CommitQuery* is called – the latter query releasing the table to be read and written. For more details on how these queries are used and called, the part of the implementation is provided below:

```
326     using (SqlConnection connection = new SqlConnection(connectionString))
327     {
328         connection.Open();
329
330         SqlCommand command1 = new SqlCommand(ProductQuery, connection);
331         SqlDataReader reader1 = command1.ExecuteReader();
332         try
333         {
334             while (reader1.Read())
335             {
336                 Product product = new Product();
337
338                 product.ProductId = int.Parse(reader1["ProductId"].ToString());
339                 product.Name = reader1["Name"].ToString();
340                 product.Description = reader1["Description"].ToString();
341                 product.ImageRef = reader1["ImageRef"].ToString();
342                 product.Price = int.Parse(reader1["Price"].ToString());
343                 product.Inventory = int.Parse(reader1["Inventory"].ToString());
344
345                 products.Add(product);
346             }
347         }
348         finally
349         {
350             reader1.Close();
351         }
352
353         if (CanBeConfirmed(orderDto.OrderItems, products))
354         {
355             for (int i = 0; i < orderDto.OrderItems.Count; i++)
356             {
357                 int productId = orderDto.OrderItems[i].product.ProductId;
358                 for (int j = 0; j < products.Count; j++)
359                 {
360                     if (products[j].ProductId == productId)
361                     {
362                         int NewInventory = products[j].Inventory - orderDto.OrderItems[i].Quantity;
363                         string UpdateInventoryQuery = " UPDATE dbo.Products" +
364                                                       " SET dbo.Products.Inventory = " + NewInventory +
365                                                       " WHERE Productid = " + productId;
366                         SqlCommand command2 = new SqlCommand(UpdateInventoryQuery, connection);
367                         SqlDataReader reader2 = command2.ExecuteReader();
368                         try { while (reader2.Read()) { } }
369                         finally { reader2.Close(); }
370                     }
371                 }
372             }
373
374
375             SqlCommand command3 = new SqlCommand(CommitQuery, connection);
```

Figure 71 Post new Order order – part 1

It is worth outlining that this development bottleneck exposed one of the *weaknesses* of the *Entity Framework* which opens and closes immediately the connection with the database server as the request is consumed resulting in an inability to use transactions unless a custom made implementation is provided as it is the case here. Therefore, as it can be seen in *Figure 71– line 328 – 330* the *SqlConnection* is manually opened and is kept open until the transaction is resolved which happens in *line 375* through a new *SqlCommand* that commits the transaction. In-between, the code reads the *Products* input through the *reader1* variable. If the order of a Customer can be confirmed, the logic from *lines 353 – 372* will be applied resulting ultimately in an update of the inventory while the transaction is still active: this means that the request waiting in the queue can perform with the updated inventory for the product. *Nonetheless, this is a chance for the reader to understand how the inventory is used to the benefit of a reliably implemented system and not only as a use case specific for the administrator's management of orders.*

```
376             SqlDataReader reader3 = command3.ExecuteReader();
377             try { while (reader3.Read()) { } }
378             finally { reader3.Close(); }
379             connection.Close();
380
381
382             productAuditIds = UpdateProductInventories(orderDto.OrderItems);
383
384             int NewOrderId = PlaceNewOrder(userId, orderDto.TotalPrice, "CONFIRMED");
385
386             PlaceOrderItems(productAuditIds, orderDto.OrderItems, NewOrderId);
387             PlaceShippingDetails(orderDto.ShippingDetailsModel, NewOrderId);
388
389             Thread t = new Thread(() =>
390                 EmailController.SendEmailAboutStatusOfOrder(NewOrderId, orderDto.ShippingDetailsModel.City, userEmail, "CONFIRMED", true));
391             t.Start();
392         }
393         else
394         {
395
396             SqlCommand command3 = new SqlCommand(CommitQuery, connection);
397
398             SqlDataReader reader3 = command3.ExecuteReader();
399
400             try { while (reader3.Read()) { } }
401             finally { reader3.Close(); }
402
403             connection.Close();
404
405             productAuditIds = DontUpdateProductInventories(orderDto.OrderItems);
406
407             int NewOrderId = PlaceNewOrder(userId, orderDto.TotalPrice, "PENDING");
408             PlaceOrderItems(productAuditIds, orderDto.OrderItems, NewOrderId);
409             PlaceShippingDetails(orderDto.ShippingDetailsModel, NewOrderId);
410
411             Thread t = new Thread(() =>
412                 EmailController.SendEmailAboutStatusOfOrder(NewOrderId, orderDto.ShippingDetailsModel.City, userEmail, "PENDING", true));
413             t.Start();
414         }
415     }
```

Figure 72 Post new Order - part 2

Having an order from the customer that can be processed, the *Products_audit table* for the specific *Id* is updated with the new inventory and, as it can be seen in *Figure 72, lines 389-391,* an email is sent to the customer with the confirmation for the order. Otherwise *(if CanBeConfirmed method returns false)*, the transaction will be immediately closed since there are is not enough inventory for the product and an email will be sent to the customer with a *PENDING status* for the order until the administrator addresses the situation by manually adding more inventory for the product as part of the *Manage order* functionality.

5.3.3 Audit Solution

To a large extent related to the *Place an order* functionality, the audit solution became a necessity when the developing team realized that this feature would result in a constraint between the *OrderItems* and *Products* table. Consequently, this brought forward issues regarding the consistency of data and further interfere with the functionality of deleting products from the *data model.*

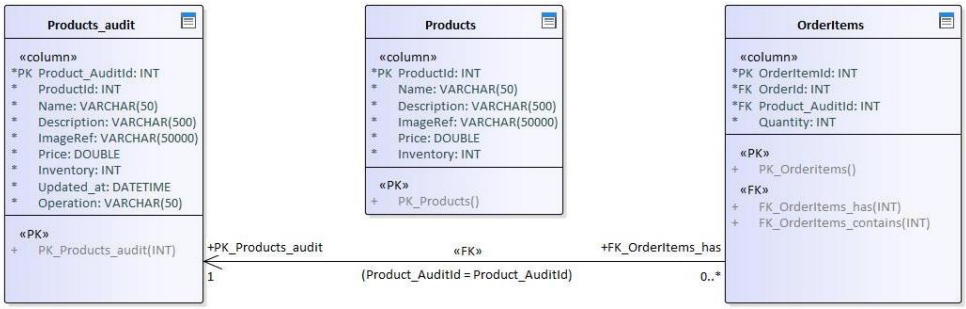


Figure 73 ER Diagram for Audit Solution

Furthermore, despite the fact that an alternative solution was discussed which would have added *isAvailable* or *isDeleted* to the *Products* table, this would have still resulted inconsistency issues when a product would be updated by an administrator since the properties of the updated product will be the ones displayed under the *My Orders UI* of the customer. The solution provided graphically by *Figure 73* was the one considered most suitable. Providing the audit as a way to keep track of all products regardless of the transformations that occurred is only beneficial to the reliability of the user experience.

Therefore, with the actual implementation that can be seen in *Figure* in the form of *SQL code*, each time the administrator is altering anything in the *Products table,* a trigger is called which replicates the row that was altered from *Products* into the *Products_audit table*, including the *timestamp* for when the change occurred and the nature of the operation *(inserted, updated, deleted)*. Consequently, every time an order is placed, the *OrderItem table* is referencing the *Products_audit table* through a *foreign key,* thus setting the actual *Products_auditId* which represents the product at the time of purchase. The same implementation that can be seen in both *Figure 74* was used for the *DonationProgram case.*

```
CREATE TRIGGER [dbo].[Products_AuditTrigger]
ON [dbo].[Products]
FOR INSERT, UPDATE , DELETE
AS
IF EXISTS ( SELECT 0 FROM Deleted )
        BEGIN
            IF EXISTS ( SELECT 0 FROM Inserted )
                BEGIN
                    INSERT  INTO dbo.Products_audit
                            (
                              ProductId ,
                              Name ,
                              Description ,
                              ImageRef ,
                              Price ,
                              Inventory,
                              Updated_at,
                              Operation
                            )
                            SELECT  D.Productid ,
                                    D.Name ,
                                    D.Description ,
                                    D.ImageRef ,
                                    D.Price ,
                                    Inventory,
                                    GETDATE() ,
                                    'UPDATED'
                            FROM    inserted D
                END
            ELSE
                BEGIN
                    INSERT  INTO dbo.Products_audit
                            ( ProductId ,
                              Name ,
                              Description ,
                              ImageRef ,
                              Price ,
                              Inventory,
                              Updated_at ,
                              Operation
                            )
                            SELECT  D.Productid ,
                                    D.Name ,
                                    D.Description ,
                                    D.ImageRef ,
                                    D.Price ,
                                    Inventory,
                                    GETDATE() ,
                                    'DELETED'
                            FROM    Deleted D
                END
        ELSE
            BEGIN
                INSERT  INTO dbo.Products_audit
                        (
                          ProductId ,
                          Name ,
                          Description ,
                          ImageRef ,
                          Price ,
                          Inventory,
                          Updated_at ,
                          Operation
                        )
                        SELECT  I.Productid ,
                                I.Name ,
                                I.Description ,
                                I.ImageRef ,
                                I.Price ,
                                I.Inventory,
                                GETDATE() ,
                                'INSERTED'
                        FROM    Inserted I
            END
GO
ALTER TABLE [dbo].[Products] ENABLE TRIGGER [Products_AuditTrigger]
GO
```

Figure 74 Triggers Sql implementation

6 Test

This section will aim to offer an extensive overview of how testing was approached both during the development of the Rotaract E-shop web application and after the *Implementation model* was completed. Testing the solution implied the usage of both *black* and *white box* testing and the upcoming sections will focus on *what* and *how* different *testing technologies* were used specifically for the *presentation tier,* respectively the *business logic tier.* Nonetheless, it must be mentioned that the following *testing methodologies* were applied:

- **Unit testing** – employed mostly for the back-end for testing the public APIs of the *Controllers* package. The methodology was necessary to assess the behavior of the endpoints in isolation from other concerns.
- **Integration testing** – used extensively for the client application to test the interaction between multiple components with routing and services included. This approach was also used in several cases for testing the back-end endpoints interaction with an in-memory mocked database.
- **System testing –** this approach was employed by the developing team to analyze how the requirements and subsequent Use Cases were fulfilled and realized by the implemented solution.

6.1 Front-end Testing

To offer a proper introduction to this section requires a brief overview of the technologies used for testing the client application. In the case of this project, ***Jasmine*** and ***Karma*** were selected for this purpose because of the ease of writing the tests with accessible frameworks that can be installed through Angular CLI. Whereas *Jasmine* is an open-source behavior-driven testing framework which provides methods such as: *it* – for test declaration, *describe* – to include a suite of tests or *expect* – to assess the validity of the results, *Karma* is the test runner tool which creates a browser instance to visualize the expected results in a way that refreshes automatically after each minor change in the tests. (*Unit Testing Angular 10/9/8 Application with Jasmine & Karma - positronX.io*)

Figure 75 includes all of the integrated tests written in Jasmine for the main selected Use Cases and their successful results depicted by Karma in the web browser. As it can seen, the following components were subject for thorough testing: *Donation Program Form, Donation Programs Overview, Orders, Register* and *Place Order.* The developing team focused entirely on integrated tests rather than unit tests (which could be carried also through inspect or UI testing) because it was more relevant to analyze the behavior of the components when different services and validators were included.

Figure 75 Karma Test results

For the purpose of presentation, the integrated tests for the *Donation Program Overview* and *Place Order* components were selected for a more detailed overview.

6.1.1 Donation Program Overview Tests

Creating the integrated tests required a certain setup that is part of a generalized approach for all the other components tested. Therefore, as it can be seen in *Figure 76,* various constant variables required to be declared such as the *Jasmine's Spy Objects* that recreate the router or services functionality *(lines 15-19)*, the header to check and simulate the response from the server and several *donation programs* objects onto which the services should perform API calls *(lines 21 - 62)*. Additionally, *lines 64 – 73* depict how the test is described with a suitable name and define few variables and function declarations to mark the test *fixture* and how changes are handled throughout the simulations.

```
15  const routerSpy = jasmine.createSpyObj( baseName: 'Router', methodNames: ['navigate']);
16  const resourcesServiceSpy = jasmine.createSpyObj( baseName: 'ResourcesService',
17    methodNames: ['deleteDonationProgram', 'getDonationPrograms']);
18  const modalServiceSpy = jasmine.createSpyObj( baseName: 'NgbModal', methodNames: ['open']);
19  const toastrServicespy = jasmine.createSpyObj( baseName: 'ToastrService', methodNames: ['error', 'success']);
20
21  const header = new HttpHeaders( headers: {
22    'x-pagination' : JSON.stringify( value: {totalPages: 9})
23  });
24  const donationPrograms = { body: [...], headers: header};
62  const route = ({data: of( args: { donationPrograms})} as any);
63
64  describe( description: 'Donation Programs Overview Component Integrated Test', specDefinitions: () => {
65    let fixture: ComponentFixture<DonationsProgramsOverviewComponent>;
66
67    function advance(f: ComponentFixture<any>) {
68      tick();
69      f.detectChanges();
70    }
71
72    let geDonationProgramsSpy;
73    let deleteDonationProgramSpy;
```

Figure 76 Donation Programs Overview Component simulated services and variables

As it can be seen in the figures below, the *Describe method* include the entire test case with all the subtests. *Figure 77* emphasizes on the method *beforeEach* which is called before each test and inside of which the *TestBed* is configured with all necessary *imports, providers and declarations* in order to compile the simulated component. In *lines 95-96* it can be seen how variables declared before are used to simulate the call of *getDonationPrograms* and *deleteDonationPrograms* methods and how the *Rxjs library* is employed using *Observable of method* to capture the returned values.

```
74  beforeEach(async( fn: () => {
75    TestBed.configureTestingModule( moduleDef: {
76      imports: [
77        RouterTestingModule,
78        BrowserAnimationsModule,
79        ReactiveFormsModule,
80        SharedComponentsModule
81      ],
82      providers: [
83        {provide: ResourcesService, useValue: resourcesServiceSpy},
84        FormBuilder,
85        {provide: Router, useValue: routerSpy },
86        {provide: NgbModal, useValue: modalServiceSpy},
87        {provide: ActivatedRoute, useValue: route},
88        {provide: ToastrService, useValue: toastrServicespy}
89      ],
90      declarations: [ DonationsProgramsOverviewComponent ]
91    }).compileComponents();
92
93    fixture = TestBed.createComponent(DonationsProgramsOverviewComponent);
94    fixture.detectChanges();
95    geDonationProgramsSpy = resourcesServiceSpy.getDonationPrograms.and.returnValue(of(donationPrograms));
96    deleteDonationProgramSpy = resourcesServiceSpy.deleteDonationProgram.and.returnValue(of( args: true));
97  }));
```

Figure 77 Donation Programs Overview Component TestBed

In *Figure 78*, each test case is declared inside the *it method* as an expectation of a certain result and the *fakeAsync* method is used to simulate the test. Inside of the latter method, various elements from the UI are declared and instantiated *(such as a button with various query selectors)*, the action from the user is simulated and the *detectChanges* method is called upon the simulated fixture to recreate the functionality. In the same figure it can be seen how the

success of the test cases is assessed based on the *expect* conditions provided. Some of the test cases considered relevant for the component are also represented in *Figure 78*.

```
98  it( expectation: 'should have loaded data form activated route before initialization ', fakeAsync( fn: () => {
99    fixture.detectChanges();
100   expect(fixture.componentInstance.donationPrograms).toEqual(donationPrograms.body);
101 }));
102
103 it( expectation: 'modalService open() should be called when edit ', fakeAsync( fn: () => {
104   const button = fixture.debugElement.nativeElement.querySelector( selectors: '#edit17');
105   button.click();
106   fixture.detectChanges();
107   expect(modalServiceSpy.open).toHaveBeenCalled();
108 }));
109
110 it( expectation: 'modalService open() should be called when add ', fakeAsync( fn: () => {
111   const button = fixture.debugElement.nativeElement.querySelector( selectors: '#add');
112   button.click();
113   fixture.detectChanges();
114   expect(modalServiceSpy.open).toHaveBeenCalled();
115 }));
116
117 it( expectation: 'resourceService deleteDonationProgram(donationProgramId) should be called when delete ', fakeAsync( fn: () => {
118   const button = fixture.debugElement.nativeElement.querySelector( selectors: '#delete17');
119   button.click();
120   fixture.detectChanges();
121   advance(fixture);
122   expect(resourcesServiceSpy.deleteDonationProgram).toHaveBeenCalled();
123 }));
124 });
```

Figure 78 Donation Program Overview Component tests

6.1.2 Place Order Component Tests

A core element of the Rotaract E-shop application as described also in the Analysis section is the customer's functionality of placing an order through the system. Therefore, this functionality was considered relevant to be presented from the perspective of how it was tested inside the client application.

In *Figure 79* is depicted how the test case for the component was configured with all the required declarations of the *Jasmine's spy objects*, the mocked data for the *shipping details and the cart items* and the definition provided for the *describe* method which include the configuration and the compilation of the simulated component.

```
11  const routerSpy = jasmine.createSpyObj( baseName: 'Router',  methodNames: ['navigate']);
12  const resourcesServiceSpy = jasmine.createSpyObj( baseName: 'ResourcesService',  methodNames: ['placeOrder']);
13  const cartServiceSpy = jasmine.createSpyObj( baseName: 'CartService',  methodNames: ['cart', 'getTotal', 'emptyCart']);
14  const validShippingAddress = {Country: 'Denmark'...};
24  const cartItems = {...};
61  describe( description: 'Place Order Component Integrated Test',  specDefinitions: () => {
62    let fixture: ComponentFixture<PlaceOrderComponent>;
63
64    function updateForm(Country, City, AddressLine, CountyOrRegion, PostalCode, PhoneNumber, FirstName, LastName ) {...}
74    function advance(f: ComponentFixture<any>) {
75      tick();
76      f.detectChanges();
77    }
78    let placeOrderSpy;
79    let getCartSpy;
80    let getTotalCartSpy;
81    beforeEach(async( fn: () => {
82      TestBed.configureTestingModule( moduleDef: {
83        imports: [
84          RouterTestingModule,
85          BrowserAnimationsModule,
86          ReactiveFormsModule
87        ],
88        providers: [
89          {provide: ResourcesService, useValue: resourcesServiceSpy},
90          FormBuilder,
91          {provide: Router, useValue: routerSpy },
92          {provide: CartService, useValue: cartServiceSpy}
93        ],
94        declarations: [ PlaceOrderComponent ]
95      }).compileComponents();
96
97      fixture = TestBed.createComponent(PlaceOrderComponent);
98
99      getCartSpy = cartServiceSpy.cart.and.returnValue(cartItems);
```

Figure 79 Place an order TestBed

In *Figure 80*, the sub cases for the component integrated test are provided. Therefore, it can be seen that of special interest were how *placeOrder* service is called and when it should not be called, how the *cartService* handles the case of emptying the cart after the order is placed and how different routes are used depending on the user interaction with the client interface.

```
104  it( expectation: 'ResourceService placeOrder() should be called', fakeAsync( fn: () => {
105    updateForm(validShippingAddress.Country,
106      validShippingAddress.City,
107      validShippingAddress.AddressLine,
108      validShippingAddress.CountyOrRegion,
109      validShippingAddress.PostalCode,
110      validShippingAddress.PhoneNumber,
111      validShippingAddress.FirstName,
112      validShippingAddress.LastName);
113
114    fixture.detectChanges();
115    const button = fixture.debugElement.nativeElement.querySelector( selectors: '#pay');
116    button.click();
117    fixture.detectChanges();
118    advance(fixture);
119    expect(resourcesServiceSpy.placeOrder).toHaveBeenCalled();
120  }));
121
122  it( expectation: 'ResourceService placeOrder() should not be called if form invalid', fakeAsync( fn: () => {...}));
138
139  it( expectation: 'cartService emptyCart() should be called when order is placed', fakeAsync( fn: () => {...}));
156
157  it( expectation: 'should route to home/profile/orders when order is placed', fakeAsync( fn: () => {...}));
176
177  it( expectation: 'should route to home/catalogue if canceled', fakeAsync( fn: () => {...}));
186 });
187
```

Figure 80 Place an order Test

6.1.3 Testing during development

The *whitebox* testing approach employed by the developing team for each iteration involving a component/module implementation was the classic one of using the *Inspect* tool and *console log* to check how the methods are called and what variables or *JSON string messages* are returned by the server after a request from the client application. This method is a commonly used one and it proved to be efficient because it facilitated the process of finding solutions for various front-end related development issues.

Another method of such testing – represented in *Figure 81* – is the usage of the *Auguri* extension tool which allowed the team to visualize the content and the objects displayed in the client interface for the entire component tree of the application.

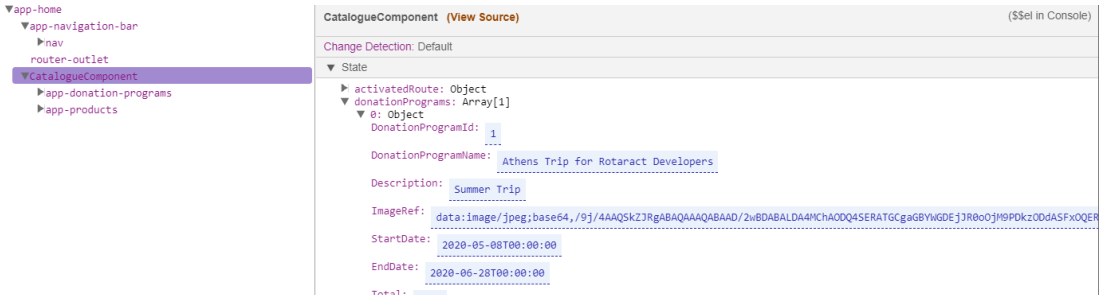


Figure 81 Auguri Extension in DevTools

6.2 Back-end Testing

The testing of the *Business logic tier* was focused more on analyzing the behavior of the API endpoints in isolation *(unit testing)* but also on integrating the mocked in-memory database for getting the full picture on how these endpoints forward the client requests to the methods of the *RotaractRepository* class which further mediate the communication with the *Entity Framework context*. In order to reproduce the tables of database with relevancy to the purpose of testing, a mocking framework tool for .NET was used: *Moq* which was installed inside the *Tests* project through the *NuGet packages manager.*

Before pursuing with more specific details for the unit and integrated tests of the back-end, an overview of all tests such as the one provided in *Figure 82* is appropriate:

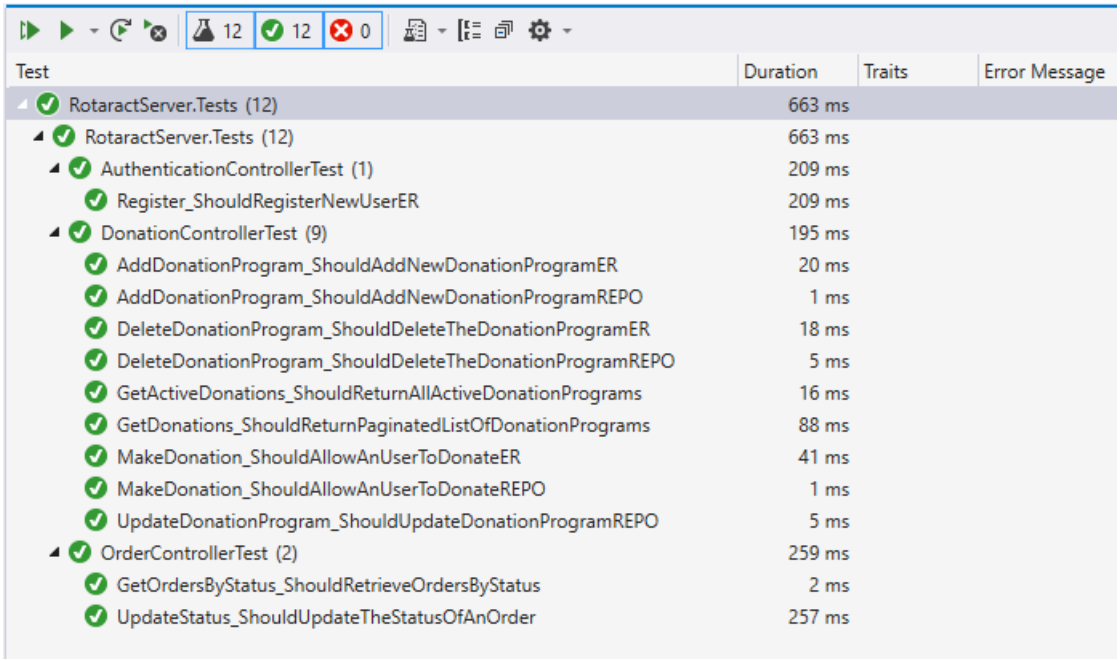


Figure 82 Integrated tests Results

Discussed in previous sections of this report, using the *Repository with Unit of work design pattern* facilitated the creation of the automated unit tests – therefore, whenever a "controller runs under a unit test class, it receives a repository that works with data stored in a way that you can easily manipulate for testing, such as an in-memory collection." (*Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10) | Microsoft Docs*) This approach will be more visible in the upcoming section where implementation details regarding the *DonationController* and the *AuthenticationController* will be provided for both unit and integrated test methods.

6.2.1 Unit Testing

This section will present how unit testing was done for the *DonationController* with regards to *adding a new donation program* (administrator feature) and *making a donation* (any user's feature).

In *Figure 83*, until *line 423*, a new variable of type *DonationProgram* is created with all the necessary properties included. Thereafter, a principle of *unit of work* is applied and a mock of the repository interface is instantiated with its object being parsed as a parameter inside the *arrange* implementation of a controller instance that takes as properties also a new request object and an *HttpConfiguration*. In the *act* implementation, the return of the *AddDonationProgram* with the given client-side donation is attributed to the result variable and thereafter an *assertion* is made regarding the returned *StatusCode* which in this case should be *HttpStatusCode.Created*.

```
399         [TestMethod]
            0 references
400         public void AddDonationProgram_ShouldAddNewDonationProgramREPO()
401         {
402             var mockDonationProgramsDto = new DonationProgramDto()
403             {
404                 DonationProgramId = 1,
405                 DonationProgramName = "EndPolio",
406                 Description = "End Polio Now!",
407                 ImageRef = "Ref",
408                 StartDate = DateTime.Now,
409                 EndDate = DateTime.Now,
410                 Total = 100
411             };
412
413             var mockExpected = new DonationProgram()
414             {
415                 DonationProgramId = mockDonationProgramsDto.DonationProgramId,
416                 DonationProgramName = mockDonationProgramsDto.DonationProgramName,
417                 Description = mockDonationProgramsDto.Description,
418                 ImageRef = mockDonationProgramsDto.ImageRef,
419                 StartDate = mockDonationProgramsDto.StartDate,
420                 EndDate = mockDonationProgramsDto.EndDate,
421                 Total = mockDonationProgramsDto.Total
422             };
423
424             var mockRepo = new Mock<IRotaractRepository>();
425
426             mockRepo.Setup(d => d.AddDonationProgram(mockExpected));
427
428             //Arrange
429             DonationController donationController = new DonationController(mockRepo.Object)
430             {
431                 Request = new System.Net.Http.HttpRequestMessage(),
432                 Configuration = new HttpConfiguration()
433             };
434
435             //Act
436             var result = donationController.AddDonationProgram(mockDonationProgramsDto);
437
438             //Assert
439             Assert.AreEqual(HttpStatusCode.Created, result.StatusCode);
440         }
```

Figure 83 Donation Program Controller mocks

A similar unit testing approach was used to simulate the functionality of a user making a donation to a program and more details are provided in *Figure 84*. In this case, simulating a donation by a user requires the actual donation and an amount that is donated. The mocked repository setup includes this object alongside with the *userId* as parameters for the repository *Donate* method and the mocked object is thereafter parsed as a parameter for the *Controller* instance. The result of the *MakeDonation* method called by the latter instance is then asserted and a successful call implies that the returned *StatusCode* is the *Accepted* one.

```
220         [TestMethod]
            0 references
221         public void MakeDonation_ShouldAllowAnUserToDonateREPO()
222         {
223             //declare and init
224             var mockDonationProgram = new DonationProgram
225             {
226                 DonationProgramId = 1,
227                 DonationProgramName = "One",
228                 Description = "Maxim one",
229                 ImageRef = "An image ref",
230                 StartDate = DateTime.Now,
231                 EndDate = DateTime.Now,
232                 Total = 111
233             };
234             const int AmountToDonate = 20;
235             const int UserID = 1234;
236
237             var mockExpected = new DonationDto()
238             {
239                 donationProgram = mockDonationProgram,
240                 amount = AmountToDonate
241             };
242
243             var mockRepo = new Mock<IRotaractRepository>();
244
245             mockRepo.Setup(d => d.Donate(mockExpected, UserID));
246
247             //Arrange
248             DonationController donationController = new DonationController(mockRepo.Object)
249             {
250                 Request = new System.Net.Http.HttpRequestMessage(),
251                 Configuration = new HttpConfiguration()
252             };
253
254             //Act
255             var result = donationController.MakeDonation(mockExpected, UserID);
256
257             //Assert
258             Assert.AreEqual(HttpStatusCode.Accepted, result.StatusCode);
259         }
260
```

Figure 84 MakeDonation Test

6.2.2 Integrated Testing

This section presents how integrated testing methods were defined for *adding a new donation* and *registering a new user* by employing the mocked persistence layer into the context.

```
18      [TestClass]
        0 references
19      public class AuthenticationControllerTest
20      {
21          [TestMethod]
            0 references
22          public void Register_ShouldRegisterNewUserER()
23          {
24              var mockUser = new List<User>...;
38
39              var districtNo = new List<DistrictNo>...;
50
51              var context = new Mock<RotaractServrerContext>();
52
53              var mockDbSet = new Mock<DbSet<User>>();
54              mockDbSet.Setup(s => s.FindAsync(It.IsAny<int>())).Returns(Task.FromResult(new User()));
55
56              mockDbSet.As<IQueryable<User>>()
57                  .Setup(m => m.Provider)
58                  .Returns(mockUser.AsQueryable().Provider);
59              mockDbSet.As<IQueryable<User>>()
60                  .Setup(m => m.Expression)
61                  .Returns(mockUser.AsQueryable().Expression);
62              mockDbSet.As<IQueryable<User>>()
63                  .Setup(m => m.ElementType)
64                  .Returns(mockUser.AsQueryable().ElementType);
65              mockDbSet.As<IQueryable<User>>()
66                  .Setup(m => m.GetEnumerator())
67                  .Returns(mockUser.GetEnumerator());
68
69              mockDbSet.Setup(m => m.Add(It.IsAny<User>()))
70                  .Callback<User>((entity) => mockUser.Add(entity));
71              context.Setup(c => c.Users)
72                  .Returns(mockDbSet.Object);
```

Figure 85 Variables simulation for testing Register

As it can be seen in *Figure 85,* variables such as *mockUser* or *districtNo* were instantiated as lists in order to be able to use them as objects of the *IQueryable* interface. In *line 51,* a context variable is declared to contain the mocked *RotaractServerContext* and the code that follows: *lines 53 – 72,* defines the procedure by which an in-memory database table is created for the *User* entity object. Thereafter, the setup involves adding rows through *callbacks* to the mocked data context, the latter being called upon to return these rows.

```
77              mockDbSet2.As<IQueryable<DistrictNo>>()
78                  .Setup(m => m.Provider)
79                  .Returns(mockUser.AsQueryable().Provider);
80              mockDbSet2.As<IQueryable<DistrictNo>>()
81                  .Setup(m => m.Expression)
82                  .Returns(mockUser.AsQueryable().Expression);
83              mockDbSet2.As<IQueryable<DistrictNo>>()
84                  .Setup(m => m.ElementType)
85                  .Returns(mockUser.AsQueryable().ElementType);
86              mockDbSet2.As<IQueryable<DistrictNo>>()
87                  .Setup(m => m.GetEnumerator())
88                  .Returns(districtNo.GetEnumerator());
89
90              mockDbSet2.Setup(m => m.Add(It.IsAny<DistrictNo>()))
91                  .Callback<DistrictNo>((entity) => districtNo.Add(entity));
92              context.Setup(c => c.DistrictNos)
93                  .Returns(mockDbSet2.Object);
94              //Arrange
95              var repoDataAccess = new RotaractRepository(context.Object);
96
97              var userExpected = new User...;
108             //Act
109             repoDataAccess.AddUser(userExpected);
110
111             //Assert
112             Assert.IsTrue(mockUser.Any(u => u.DistrictNumber == 1));
113             Assert.AreEqual(mockUser[0].UserId, userExpected.UserId);
114         }
115     }
```

Figure 86 Authentication Controller Integrated Test

In *Figure 86,* it can be seen that a new mock *DbSet* is created for the *DistrictNo* because of the actual constraints required by the real implementation. The procedure of creating this table is similar to the one presented above, however, the mocked user is defined throughout this new mocked set implementation as a simulated constraint to the *DistrictNo* table.

After these initial steps, a new variable *repoDataAccess* is declared to hold a reference to the *RotaractRepository* implementations *(which holds the context object as a parameter)*. Thereafter, an expected user is created and a method call is made upon the *repoDataAccess* variable with the user parsed as a parameter. Ultimately *(as it can be seen in line 113)* an assertion is made to check if the mocked user' Id from the context object is equal to the one of the *expected user* which would prove that the value was added to the in-memory database.

Another example of an integrated test method can be seen in *Figure 87* in the case of adding a donation program to the in-memory database. In this case, most of the testing procedure involves similar steps apart from when the assertion is made which checks for particular properties of the added donation program.

```
442         [TestMethod]
            0 references
443         public void AddDonationProgram_ShouldAddNewDonationProgramER()
444         {
445             var mockDonationProgramList = new List<DonationProgram>()...;
468
469             var context = new Mock<RotaractServrerContext>();
470
471             #region MockDbSetup DonationProgram
472             var mockDbSet = new Mock<DbSet<DonationProgram>>();
473             mockDbSet.Setup(s => s.FindAsync(It.IsAny<int>())).Returns(Task.FromResult(new DonationProgram()));
474
475             mockDbSet.As<IQueryable<DonationProgram>>()
476                 .Setup(m => m.Provider)
477                 .Returns(mockDonationProgramList.AsQueryable().Provider);
478             mockDbSet.As<IQueryable<DonationProgram>>()
479                 .Setup(m => m.Expression)
480                 .Returns(mockDonationProgramList.AsQueryable().Expression);
481             mockDbSet.As<IQueryable<DonationProgram>>()
482                 .Setup(m => m.ElementType)
483                 .Returns(mockDonationProgramList.AsQueryable().ElementType);
484             mockDbSet.As<IQueryable<DonationProgram>>()
485                 .Setup(m => m.GetEnumerator())
486                 .Returns(mockDonationProgramList.GetEnumerator());
487
488             mockDbSet.Setup(m => m.Add(It.IsAny<DonationProgram>()))
489                 .Callback<DonationProgram>((entity) => mockDonationProgramList.Add(entity));
490             context.Setup(c => c.DonationPrograms)
491                 .Returns(mockDbSet.Object);
492             #endregion
493
494             var repoDonationDataAccess = new RotaractRepository(context.Object);
495
496             var donationProgram = new DonationProgram...;
506
507             repoDonationDataAccess.AddDonationProgram(donationProgram);
508
509             //Asserts
510             Assert.IsTrue(mockDonationProgramList.Any(x => x.DonationProgramId == 3));
511             Assert.IsTrue(mockDonationProgramList.Any(x => x.DonationProgramName.Equals("Three")));
512             Assert.IsTrue(mockDonationProgramList.Any(x => x.Total == 0));
513         }
```

Figure 87 Add Donation Program Integrated Test

6.2.3 Endpoints Testing using Swagger

As discussed in previous sections of the report, Swagger was an important tool that was used throughout the development that allowed the system's endpoints to be tested remotely especially when errors occurred and their origin was unclear. The benefit that this tool provided was to visualize through an easy to use interface most of the elements defining the communication between the client and the server such as the *Request URL, Response Body, Response Code or the Headers.* Additionally, Swagger provided easy access to the tokens required by most of the endpoints for authorization and this tool could remotely recreate functionality that could only be done in the client side such as adding, deleting, editing products or donation programs through *samples of json strings* which resulted in the objects being created in the UI and stored in the database. *Figure 88* offers an overview of how the server APIs were replicated through the configuration of Swagger:

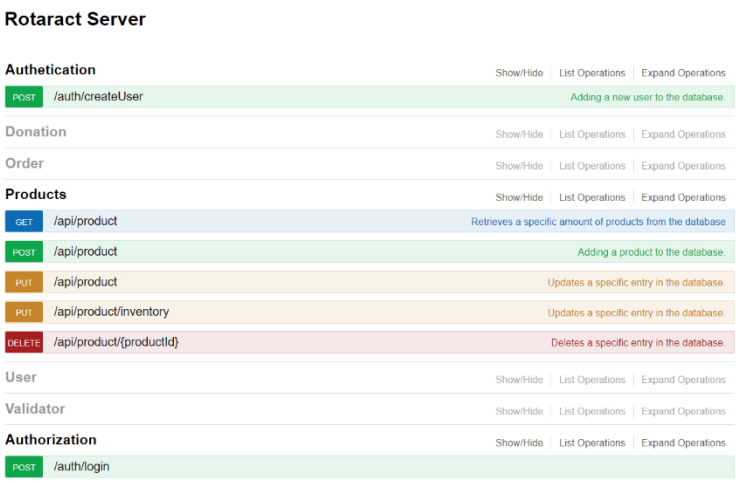


Figure 88 Swagger Documentation

More specific to the testing, Swagger improved the development pace of the system by allowing the team to make calls *GET, POST, PUT, DELETE* and offering the status code of the operation which in many cases was not just a success *STATUS 200 OK or 201 Created (as for PUT or POST),* but also *error responses from the client side such as 400 or 401* or *server error responses such as 500 or 501* (amongst the most frequently encountered). By analyzing these codes, the developing team could rapidly identify which tier was causing the error and investigate more into the problematic one until a solution was found. Besides, Swagger was also a tool supporting the tests conducted after the implementation of the application was to a large extent over as it was the case during *System testing* for which extra details are provided in the upcoming section.

6.3 System Testing

This is a level of software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified functional requirements. (*System Testing - Software Testing Fundamentals*)

6.3.1 Test Specifications

6.3.1.1 Test Case: Register as new user

***Preconditions*:** Viewer possesses a valid district number and must have a unique username and email.

Table 4 Register as new user – Test specification

Step	Action	Reaction	Result
1	User clicks on *Register*	System displays *Register form*	✓
2	User enters first name, last name and district number	System checks the district number	✓
3	User enters username	System checks username uniqueness	✓
4	User enters email	System checks email uniqueness	✓
5	User enters password and clicks register	System validates form for required fields	✓
6		System saves the account	✓
7	User is now a Customer of the application	System redirects user to login page	✓
8	User cancels registration	System redirects user to home page	✓

6.3.1.2 Test Case: Login

Preconditions: The user has an account registered (either a Customer or an Administrator one).

Table 5 Login - Test specification

Step	Action	Reaction	Result
1	User clicks on *Login*	System displays Login page	✓
2	User enters username and password	System checks if any of the required fields is left blank	✓
3	User clicks on the login button	System checks the credentials	✓
4		System displays the customer or administrator home page for the specific user	✓
5	User cancels login	System redirects user to the home page	✓

6.3.1.3 Test Case: Logout

Preconditions: The user is logged in the system.

Table 6 Logout - Test specification

Step	Action	Reaction	Result
1	User clicks on *Logout* from the account menu	System logs out the user	✓
2		System displays the home page	✓
3	User can use the application as a Viewer		✓

6.3.1.4 Test Case: Browse catalogue

Preconditions: The Viewer accesses the web shop application.

Table 7 Browse Catalogue -Test specification

Step	Action	Reaction	Result
1	User accesses the platform	System displays the home page with all the products and the donation programs	✓
2	User browses through the products catalogue and through the donation program carousel		✓

6.3.1.5 Test Case: View product details page

Preconditions: The Viewer or the registered user is on the home page of the application

Table 8 View product details page - Test specification

Step	Action	Reaction	Result
1	User clicks on any product container	System displays a modal page with more details about the product and offers the option to include the product to cart	✓
2	User clicks outside the frame of the modal page	System displays the home page again with all the products	✓

6.3.1.6 Test Case: View donation program details

Preconditions: The Viewer or the registered user is on the home page of the application

Table 9 View donation program details page - Test specification

Step	Action	Reaction	Result
1	User clicks on *More info* button link for a donation program from the carousel	System displays a modal page with more details about the donation program including the possibility for any user to donate	✓
2	User clicks on the *Close button*	System displays the home page again the donation programs	✓

6.3.1.7 Test Case: Add product to cart and remove product from cart

Preconditions: The user must hold a Customer account and must be logged in the system

Table 10 Add and remove product from cart

Step	Action	Reaction	Result
1	Customer opens the product details for a selected product	System displays a modal page with more details about the product and the *Add to cart button with the quantity option alongside*	✓
2	Customer adds the quantity and clicks the *Add to cart button*	System adds the product with specified quantity in the cart and displays a Toastr on the screen about the operation	✓
3	Customer access the Cart section of the application	System displays the product added there	✓
4	Customer clicks on *Remove item button*	System removes the product from the cart and displays a Toastr about the operation	✓

6.3.1.8 Test Case: Donate

Preconditions: The Viewer or any other user must be on the home page of the application

Table 11 Donate - Test specification

Step	Action	Reaction	Result
1	User clicks on the *Donate button* on a donation program or on the *Donate now* button from the page with more details about the program	System displays the *Support Donation Program* page	✓
2	User enters the amount to donate and clicks on *Confirm your contribution button*	System saves the amount donated for the specific program, displays a Toastr with *Thank you for your donation* and updates also the total amount in the administration overview page for the specific donation program.	✓

3	User clicks on *Close button*	System displays again the home page	✓

6.3.1.9 Test Case: Place an order

Preconditions: The user must hold a Customer account, be logged in the system and must have products in the cart to place an order with

Table 12 Place an order - Test specification

Step	Action	Reaction	Result
1	User clicks on the shopping cart icon	System displays the cart window with the items added so far and their details, quantities, price per item and the total price	✓
2	User clicks on *Place order button*	System displays the *Shipping details page*	✓
3	User enters name, address and phone number on the shipping details form	System checks if all the fields have been completed	✓
4	User chooses to proceed with the payment	System saves the order	✓
5		System clears the items from the cart corresponding to that session	✓
6		System checks if there is enough inventory for the products of the order items	✓
7		Depending on the inventory, the system sends an email to the customer with regards to the status of the order which is either: *pending* or *confirmed*	✓
8		System redirects the user to *My Orders page*	✓

6.3.1.10 Test Case: View Orders and their status

Preconditions: The user must hold a Customer account, be logged in the system and should have an order already placed in order to view it (otherwise, an empty data table is displayed)

Table 13 View Orders and their status - Test specification

Step	Action	Reaction	Result
1	User clicks the username toggle on the right side of the application page and selects *Profile*	System displays the *Profile page* with *My Orders section* being automatically selected for display	✓

2	User can view the id of the order, the status, the products, shipping details and total paid for each order in a data table view		✓

6.3.1.11 Test Case: Manage Orders with Change order status

Preconditions: The administrator must be logged in the system to be able to manage the orders.

Table 14 Manage Orders with Change order status - Test specification

Step	Action	Reaction	Result
1	Administrator chooses *Orders* from the administration menu	System displays a data table with all the orders stored in the system categorized as such: *pending, confirmed, in progress, shipping, completed and cancelled*	✓
2	Administrator can choose from each one of these categorizes to see the orders for each and to be able to manage them.	System displays the orders for each category alongside buttons with functionality under the status section and pagination of orders if necessary.	✓
3	Administrator clicks on *pending button* for a selected order.	System checks if there is enough inventory to move the order to *confirmed* and if not displays a toastr about the failure of the operation.	✓
4	For convenience, the administrator can update the inventory for the products in *Orders Overview* by pressing the + button next to the *Shipping Details section.*	System displays the *Update inventory* modal page where the user can add more products.	✓
5	If the inventory has been adjusted, by pressing *pending* again, the administrator can move the order to *confirmed.*	System moves the order to the *confirmed section* and displays a Toastr about the success of the operation.	✓
6	The administrator clicks on *Cancel button.*	System moves order to the *Cancelled* section and displays confirmation toastr.	✓
7	The administrator clicks on *confirmed button* under the section with the same name.	System moves the order to the *in progress* section and displays confirmation toastr.	✓
8	The administrator clicks on *in progress button* under the section with the same name.	System moves the order to the *Shipping* section and displays confirmation toastr.	✓
9	The administrator clicks on *shipping button* under the section with the same name.	System moves the order to the *Completed* section and displays confirmation toastr.	✓

10	The administrator clicks on the *completed button* under the section with the same name.	System displays the *not allowed toastr* because an order cannot be completed and cancelled at the same time.	✓
11	The administrator clicks on the *cancelled button* under the section with the same name.	System displays that the operation is not allowed.	✓
12		System sends an email to the customer regarding each change in order status that occurred.	✓

6.3.1.12 Test Case: Manage Products with Add, Delete, Edit product and Add inventory

Preconditions: The administrator is logged in the system to be able to manage products.

Table 15 Manage Products - Test specification

Step	Action	Reaction	Result
1	Administrator chooses *Products* from the administrator menu.	System displays a data table with all the products stored in the system, their properties, buttons for management and pagination included.	✓
2	Administrator selects a product row and clicks on *Show image*.	System displays a small window with the product image included.	✓
3	Administrator clicks on the *recycle bin icon button* under the *Actions* section.	System removes the product from the data table and from the home page catalogue.	✓
4	Administrator clicks on the *blue edit icon button*.	System displays the *Update (edit) product* page for the selected product.	✓
5	Administrator updates the desired fields and clicks *Save*	System saves the changes, closes the modal and updates the edited fields for the selected product both in data table and home page.	✓
6	Administrator clicks on the *pink + button under the Actions* section.	System displays the *Update product inventory modal page.*	✓
7	Administrator enters the desired inventory amount and clicks *Add products button.*	System saves the changes and updates the *In inventory* field with the input amount.	✓
8	Administrator clicks on the *blue + icon button* from above the data table.	System displays the modal *Add a new product* with name, description, price fields and image input frame.	✓
9	Administrator fills the fields and clicks on the *Save button.*	System checks if all the required fields are filled.	✓

10		System saves the product and adds it both in the data table and on the home catalogue page.	✓
11	Administrator clicks on the *Cancel button* for the modals described above.	System closes the specific modal and displays the *Products Overview data table.*	✓
12	Administrator chooses how many items should be displayed on the page from a toggle below the data table.	System display only the specific number of items in the data table.	✓
13	Administrator chooses the page number to be displayed.	System displays the items from the page required by the user.	✓

6.3.1.13 Test Case: Manage Donation Programs with Add, Edit and Cancel donation program

Preconditions: The administrator is logged in the system to be able to manage donation programs.

Table 16 Manage Donation Programs - Test specification

Step	Action	Reaction	Result
1	Administrator chooses *Donation Programs* from the administration menu.	System displays the data table with all the donation programs stored in the system, their status and the total amount donated.	✓
2	Administrator clicks on *Show image button*.	System displays the image of the selected donation program.	✓
3	Administrator clicks on the *blue + button.*	System displays the *Add Donation Program* modal page.	✓
4	Administrator fills the fields, chooses and image and the duration of the program and clicks on the *Save button.*	System checks if all the required fields were completed.	✓
5		System saves the newly added donation program, adds it the data table and in the home page carousel.	✓
6	Administrator clicks on the *delete icon button* from the *Actions section.*	System displays the confirmation toastr for the operation and removes the program from the data table and carousel.	✓
7	Administrator clicks on the *edit icon button* from the *Actions section.*	System displays the *Update program modal page* with the current fields which can be edited.	✓
8	Administrator updates the desired fields and clicks *Save.*	System checks for blank fields and then updates the program both in the data table and carousel.	✓

9		If the user inputs a duration period that is before the current day, the data table displays *Inactive status* for the program and removes the program from home page carousel.	✓
10	Administrator selects from the toggle how many items per page to be displayed.	System displays only the number of items required by the user.	✓
11	Administrator selects the page to be displayed.	System displays the page selected.	✓
12	Administrator clicks on *Cancel button* for all the above described modal pages.	System displays the *overview data table.*	✓
13		System displays confirmation or warning toastr labels for all the operations.	✓

6.3.2 System testing discussion

The procedure regarding system testing involved the entire developing team and can therefore be considered an internal procedure. The purpose of the section is to provide the reader of the report with a clear overview of how the *Rotaract E-shop web application* not only fulfill most of the functional requirements from *Section 2.1.1* but also the *Use Cases* deducted from these requirements *(artifacts of the UseCase diagram)* and their description.

The approach was to have a thoroughly made test of the behavior from the deployed *angular application* from the starting point when a viewer registered the account and moving forward with all the functionalities specific for both: the customer and the administrator. As it can be seen from consulting also the *Appendix C* section, the team intended as much as possible to align the system testing with the realization of the use cases – however, it is important to notice from the *action – reaction for each test case* how the development of the final solution produced minor variations for some of the use cases which are normal for the software development lifecycle and were intended to benefit the user experience. Besides this, if minor issues were found for mandatory to have requirements *(for example: that an administrator could not delete a product)* and if they could have been subject to rapid fixes *(as it was the case in this only regard which needed a quick solution), Swagger* was used to map the error and solve the problem *(which in this case was a minor issue in the back-end side)*. Therefore, system testing was also an additional method to ensure the reliability of the application.

All in all, after the internal system testing was holistically conducted, the conclusion reached is that the Rotaract E-shop system is a fully functional one at least when it comes to the *must and should requirements from section 2.1.1*. The only requirements that were not fulfilled are the following which also have a *low priority*:

- The system ***could*** provide the registered user with the possibility to recover access to the account in the case of forgetting the password.
- The system ***could*** allow the customer to add a product to a wish list.

7 Results and Discussion

As it was thoroughly presented across this report and also considering the outcomes of the system testing, the *Rotaract E-shop system* developed achieved more than satisfactory results mainly because all of the must-have and even some should-have requirements were covered and successfully implemented. Therefore, a user of the system – focusing mainly on either customer or administrator – is able to utilize the web application in a manner consistent with the initially planned functionalities described by the *Use Cases*. Consequently, if the discussion around the results of the implementation regards the consistency aspect, the solution achieved is one that is aligned with the premises of a fully functional e-commerce application taking into consideration also the predefined delimitations made to the scope of this project.

Looking briefly at the solution from a technical perspective, the application was built on the skeleton of a three-tier architecture adapted to the needs of e-commerce in which the presentation tier is the client-side implemented in Angular, the business logic tier is a server-side implemented using the .NET Framework and the data tier is a database that utilizes both the Entity Framework and the MSSQL management system for storing and altering data. For the communication between the client-side and the server-side, the solution makes use of the RESTful API approach and the subsequent methods: *get, delete, post and put* to process requests from the client and sent back responses to it from the server in the form of *HttpResponseMessages.* The discussion around some of the more interesting technical aspects for each tier would include how the client-side implements components using *ngmodals* or how *observables and validations* are used to process responses from the server, how the server-side implements the endpoints, paginations or patterns of abstraction or how the data tier was implemented using queries, an audit solution or transactions for concurrency control. Results for this discussion are covered throughout this report.

To offer a sum up for this section would probably mean to raise the most challenging dilemma: if the results have been achieved correctly using all of the tools, methodologies or programming techniques available or implied. For this possible discussion, the developing team would have the honest answer that the aim was towards that and that the knowledge and experience gathered throughout the university were to a large extent applied to achieve just that. However, the team will bring an argument on its favor for this discussion which illustrates how the audit inspect tool regards the implemented solution – *Figure 89.*

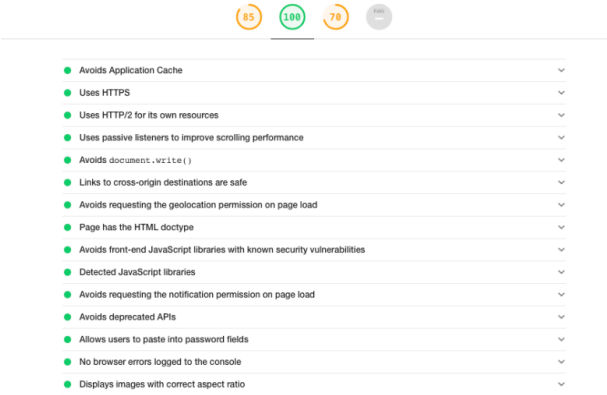


Figure 89 Best Practices Audit Results

8 Conclusions

The *Rotaract E-shop web application* is a solution that managed to successfully achieve all of the high-priority functional requirements and some of the non-functional ones especially with regards to performance or response time for the pages. Besides, the application covered a rather large number of features in a short development time for which best practices implementation approaches were used. Therefore, one of the first conclusions that can be reached is that the effort and dedication implied by developing this solution were more than reasonably in accordance with what was actually demanded by a project of such magnitude.

With regards to the stakeholders demands, the web application was implemented to cover most of the functionalities required by a classic e-commerce solution. In a nutshell, a customer is able to place an order with items added in a cart from a home catalogue page while also providing the shipping details for the order and this user can see the placed orders and receive updates about their progress even by email. Besides, an administrator is able provide the necessary background management to maintain and update the elements of the application through the in-built tool to which a customer does not have access by virtue of the authorization delimitations. Additionally, the developing team opted for extra functionality tailored made to the needs of an international nonprofit organization such as the inclusion of donation programs and donate feature that allows the application to stand out when compared to alternatives.

As per the problem statement that the project tried to solve, the application is certainly a tool through the usage of which ERIC will be able to transform the supply chain model from a push to a pull strategy. Despite the delimitations applied to the scope of this project and besides the features mentioned in the previous paragraph, the users of the application are able to request products with specific amounts through orders so that the Board can have an overview of what was ordered and what actions shall be further taken. For this, the inventory solution is an important addition that offers them more control and also covers one of the few ERIC's requests.

Finally, the developing team is to a great extent satisfied with the resulted solution. Forward-thinking to the time when the Rotaract E-shop web application will be deployed to production, the solution will be used by thousands of members across Europe and inner of itself this will be an achievement and a good start too for the future path as engineers.

9 Project Future

From the very beginning it has to be mentioned that this project is sure to have a future outside of the scope of the bachelor since this was agreed with the stakeholders during one of the meeting with them. Therefore, the solution that this project provided can be considered just a blueprint for the later fully functional e-commerce solution that will – at the minimum – be deployed to the server provided by ERIC after the payment and invoicing solutions using a 3rd party system *(Mollie)* will be integrated. Consequently, two of the main additions to the current implementation will be that a user will be able to donate to a donation program using a secured payment method and that the customer will be able – before placing an order – to securely pay for it. These further implementations will happen on a timeline that has still not been fully agreed with ERIC.

Other important additions to this solution would be represented by upgrading the security of the application to a level required by an organization with such profile by using encryption techniques. Additionally, the system can be appended with the *Forgot password functionality* (for which an endpoint is already created and the email generation technique is already in use) and with the feature of adding products to a wish list (which is rather unlikely considering the limited amount of products that ERIC currently has). Moreover, the administration page can extend its functionality with retrieving statistics about donation and products or the catalogue page can have a filtered-based search according to certain criteria for the displayed products.

Overall, the most important future for this project is that the *Rotaract E-shop web application* will be used in production by thousands of users and that it will provide added value to an organization that is involved in many projects worldwide to help develop leadership and communities.

10 Sources of Information

E-commerce, U. and Models, B. (2019) 'Chapter 1 UNDERSTANDING E-COMMERCE', pp. 13–30.

Larman, C. (2004) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, *Analysis*. doi: 10.1016/j.nec.2006.05.008.

[N]Varchar(4000) and Performance – SQLServerCentral [Online]
Available at: https://www.sqlservercentral.com/forums/topic/nvarchar4000-and-performance?fbclid=IwAR3VYdNZtUwtC-AOwq_i8B5g2drxgICDGC6FbVs1mJk-2yYKOqH0NK-f1pY

5 Benefits of a 3-Tier Architecture - Izenda [Online]
Available at: https://www.izenda.com/5-benefits-3-tier-architecture/

Angular - Dependency injection in Angular [Online]
Available at: https://angular.io/guide/dependency-injection

Angular - Introduction to Angular concepts [Online]
Available at: https://angular.io/guide/architecture

Angular - Types of feature modules [Online]
Available at: https://angular.io/guide/module-types

Angular powered Bootstrap [Online]
Available at: https://ng-bootstrap.github.io/#/home

Benefits of Entity Framework [Online]
Available at: https://www.cmarix.com/benefits-of-entity-framework/

Bootstrap (front-end framework) - Wikipedia [Online]
Available at: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

Brand Identity – Rotaract Europe [Online]
Available at: http://rotaracteurope.eu/resources/eric-visual-identity/

Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10) | Microsoft Docs [Online]
Available at: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application

Difference between .NET Framework 4.6, .Net Native and .Net Core - Stack Overflow [Online]
Available at: https://stackoverflow.com/questions/29609993/difference-between-net-framework-4-6-net-native-and-net-core

Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10) | Microsoft Docs [Online]
Available at: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Introduction to Angular Dependency Injection - TekTutorialsHub [Online]
Available at: https://www.tektutorialshub.com/angular/angular-dependency-injection/

Inversion of Control Containers and the Dependency Injection pattern [Online]
Available at: https://www.martinfowler.com/articles/injection.html

Larman, C. (2001) 'Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)', p. 656. [Online]
Available at: http://www.amazon.com/Applying-UML-Patterns-Introduction-Object-Oriented/dp/0130925691.

Rotaract Europe – Since 1988 [Online]
Available at: http://rotaracteurope.eu/.

System Testing - Software Testing Fundamentals [Online]
Available at: http://softwaretestingfundamentals.com/system-testing/

Unit Testing Angular 10/9/8 Application with Jasmine & Karma - positronX.io [Online]
Available at: https://www.positronx.io/angular-unit-testing-application-with-jasmine-karma/

What is .NET Framework? Complete Architecture Tutorial [Online]
Available at: https://www.guru99.com/net-framework.html

What is Microsoft SQL Server? A definition from WhatIs.com [Online]
Available at: https://searchsqlserver.techtarget.com/definition/SQL-Server

What is REST – Learn to create timeless REST APIs [Online]
Available at: https://restfulapi.net/

11 Appendices

Appendix A – Project Description

Appendix B – System Software Requirements

Appendix C – Analysis

Appendix D – Diagrams

Appendix G – Installation Guide