Car part Warehouse

- Project documentation -

Teaching Assistant

Tufișı Radu

Student

Frăsie Alexandru

Technical University

Of Cluj-Napoca

Group: ***30433***

Academic year: ***2019 – 2020***

MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

Contents

I Project specification

This application describes a warehouse that stocks and sells car parts.

In this way, there should be 2 types of users: the **administrator** and the **customer**.

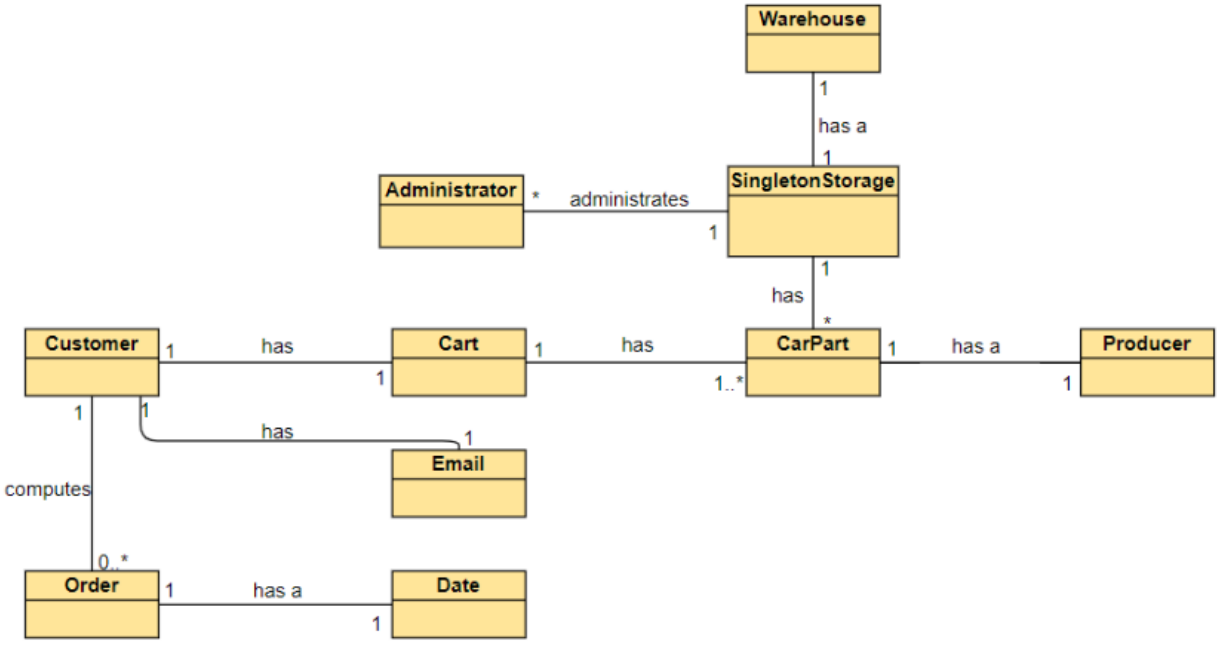
The users should be able to log in separately and the system should be able to memorize their data. If a user does not have an account, it should be able to register by pressing the *Register* button and memorize the data.

The administrator is able to administrate the storage and add/delete/update products from it at any time.

The customer, however, is able only to visualize the existing items in storage that are ready for sale. If the customer is interested in one of the products, they are able to add the product to their personal cart, which can be modified later on. At the end of the shopping session, the customer is able to place an order with the existing items in its cart. The customer will then be sent an email with the bill containing the information about the order that was just placed.

All in all, the application should work like a real-life application which has a user-friendly design and can be easily understood.

1.1 Domain Model Diagram



II Use-Case model

If the user logs in as ***administrator***:

- ✓ they are able to add new car parts
- ✓ they are able to edit existing car parts
- ✓ they are able to delete existing car parts

If the user logs in as ***customer***:

- ✓ they should be able to browse through the existing items (car parts)
- ✓ they should be able to add car parts in their personal carts
- ✓ they should be able to view their personal cart at any moment
- ✓ they should be able to compute an order

2.1 Users and stakeholders

Users:

- the administrator
- the customer

Stakeholders:

- the producers
- the warehouse manager

2.2 Use-Case identification

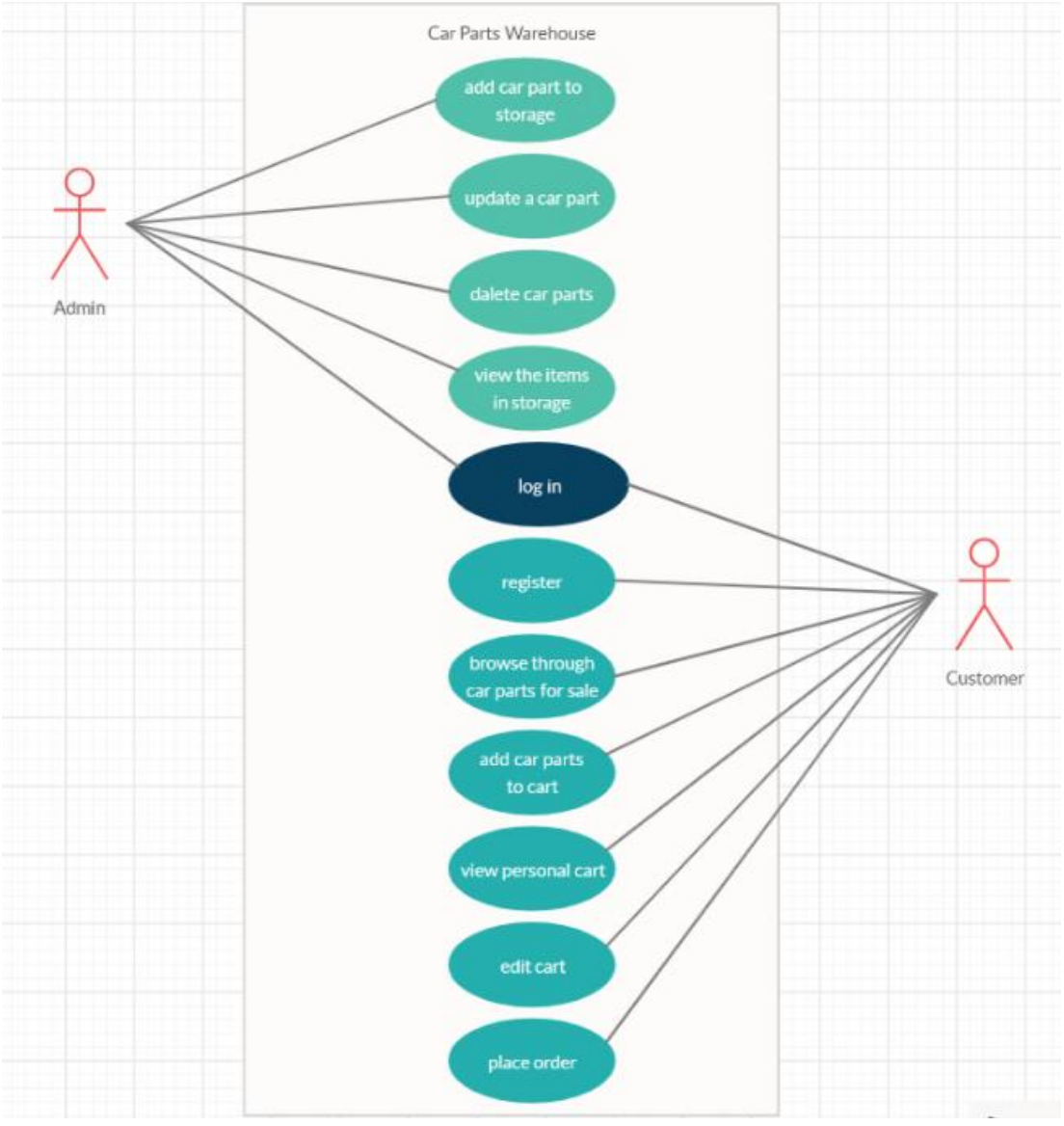
1. Use case name: *Register new customer.*
Level: *User-Goal.*
Main actor: *Customer.*
Main success scenario: *The customer is required to introduce some personal data in order to create an account. Fields like first name, last name, an email, a username and a password for the account are required. If all the introduced data is valid, then the account in created with success and the customer is now able to log in the application.*
Extension: *An exception case would be the one where a customer tries to register using a username that is already in use with a different account. Then, a message will appear telling the customer to choose another username.*

2. Use case name: *Add a car part to the cart.*
Level: *User-Goal.*
Main actor: *Customer.*
Main success scenario: *The customer is allowed to browse through the products and has the possibility to add an existing product to its personal cart, while specifying the quantity of the car part to be added. The item will now be visible in the cart section, ready for being purchased.*
Extension: *An exception would occur whenever the quantity specified is larger than the existing quantity in storage. Then, the customer will be required to change the quantity needed, displaying the maximum number of the product existing in stock.*

3. Use case name: *Compute order*
Level: *User-Goal.*
Main actor: *Customer.*
Main success scenario: *When the customer is done shopping, they have the possibility to purchase the car parts existing in their carts, by selecting the "Order" button. This will automatically create an Order, will remove the items from storage and send an email to the customer with the bill, specifying the products, the address to which the products are to be delivered and the total amount of money they should pay.*
Extension: *An exception case would be when the cart is empty, then the customer will be required to add something into the cart, or else the order will not be computed.*

4. Use case name: *Delete car parts from storage.*
Level: *User-Goal.*
Main actor: *Administrator.*
Main success scenario: *The admin is allowed to modify the storage in any way. That being said, the administrator is able to delete any existing car parts from the storage, eliminating the product from being seen/added to cart/purchased by the customers and deleting all the information about that product.*
Extension: *An exception would occur whenever a certain product is to be deleted by an administrator, but that product exists already in a customer's cart. Then, the product will also be deleted from that cart and the customer will be announced that the certain car part is no longer available for purchase.*

2.3 UML Use-Case diagram



III Architectural design

This chapter will be about the architectural design of the application. The main topics that will be discussed will be about the inter-communication between the main components of the application and how this communication is implemented. The focus will be on the key concepts that founded the architecture and on some diagrams, for better understanding of the flow.

3.1 Conceptual architecture

The Car Part Warehouse application is a *Web* application, being accessed through the World Wide Web.

It also is a *Client-Server* application, as it is modelled to be similar to an online store, which allows a customer (having the Client role) to interact with a Server. More explicitly, the application will have a client part, where the customer can visualize the functionalities of the store through a user-friendly interface and, in addition, is able to interact with it by sending requests and expecting responses from the server of the application. The other part of the application is the Server, whose purpose is to expect requests from the client, try to solve these requests by communicating with the database, and send back a response to the client, displayed on the same UI.

As mentioned before, the application has *access to a database* so that is can store and retrieve information vital for the usage of the application. So, it uses an SQL relational database, composed of multiple tables, among which there are various relationships (one-to-one, ono-to-many, many-to-many). The tables in the database are modelled 1:1 with the main classes of the application, using *Hibernate*.

In addition, the application respects the *Layered Architecture* concept and as far as the main components are concerned, the application has some model classes, some controllers and some views. That being said, it is built around the *MVC* pattern. The MVC (Model – View – Controller) architecture is used for a better structuring of the application.

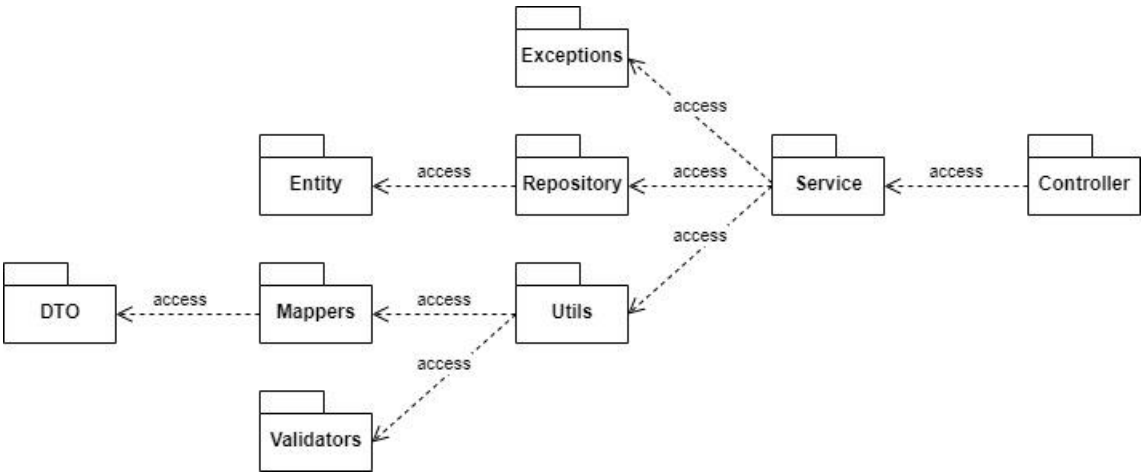
That being said, the real-life objects that we need (like the customer, or a product) are modelled into classes useful for our business logic, and in this way the model classes are created. Alongside these classes, the repositories and the service classes, which are used for business logic, represent the **model** part. The **view** part of the application is represented by the graphical interfaces which represent the data to the client. Nonetheless, the **controllers** are the ones which stay in the middle of the model and the view, linking them and maintaining a suitable transmission of requests/responses.

3.2 Package diagram

The package diagram of this application presents how the classes are organized. Every package contains classes that have a specific purpose (for example, the repositories access the database).

By using the Layered Architecture concept, the flow of transmitting the data is clear: ***DAO layer*** (the repositories that access data directly from the database), ***business layer*** (the services and all packages that validate or transform data) and the ***presentation layer*** (where the controllers receive data to be displayed on the views).

Therefore, some packages access (or import) some other packages, to maintain a logical a clean flow for the data. This is reflected in the package diagram below:



3.3 Class diagram

The class diagram is not final yet, but as far as the ***entities*** are concerned, there are classes for: Administrator, Customer, CarPart, Producer, Order, Cart and Address.

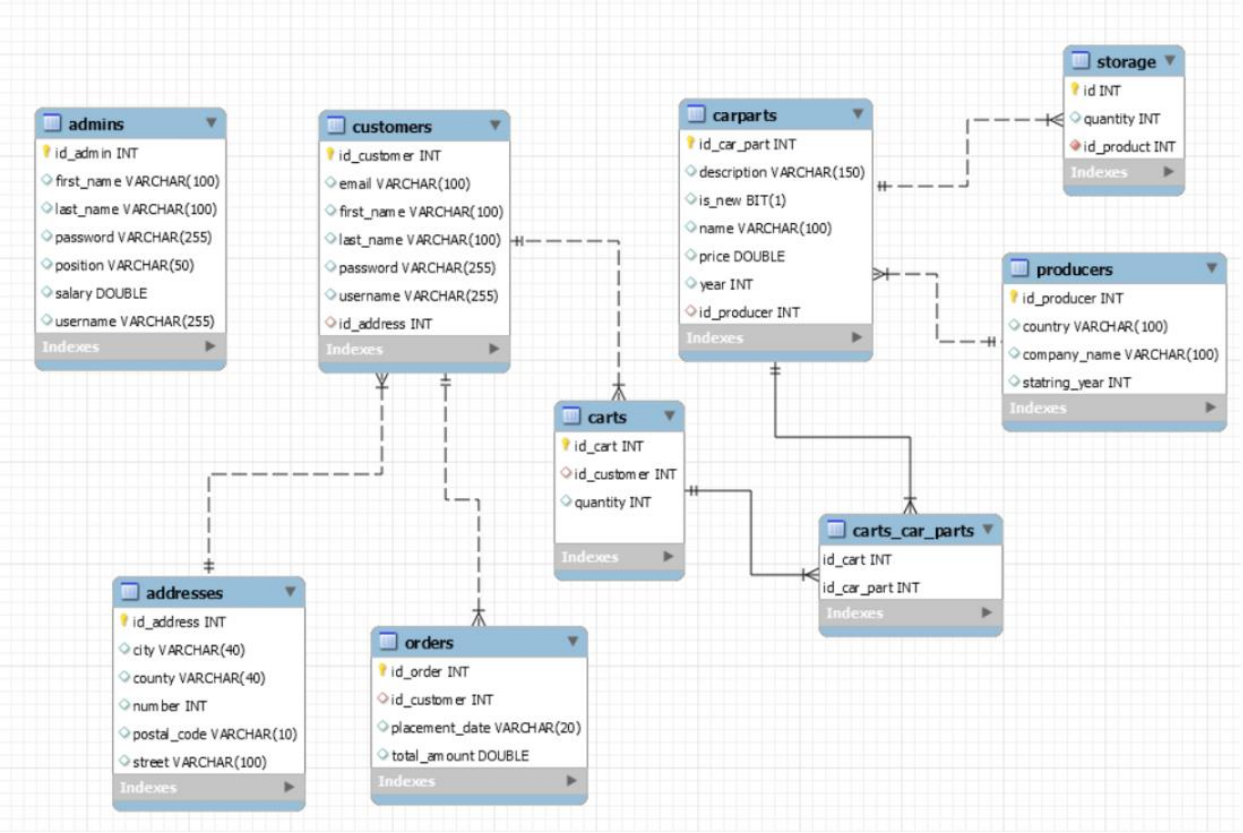
As the application is based on the Layered Architecture, for each of these classes there will be ***services*** and ***repositories***, which will be designed as interfaces.

There will exist ***controllers*** for the classes which will be involved in data transfer: CarPart, Customer, Cart, Order.

There will also exist classes for utils, like ***validators*** and ***mappers***, alongside ***dtos***.

There will be a class ***ExceptionHandler*** and some custom ***exceptions***.

3.4 Database (E-R/Data model) diagram



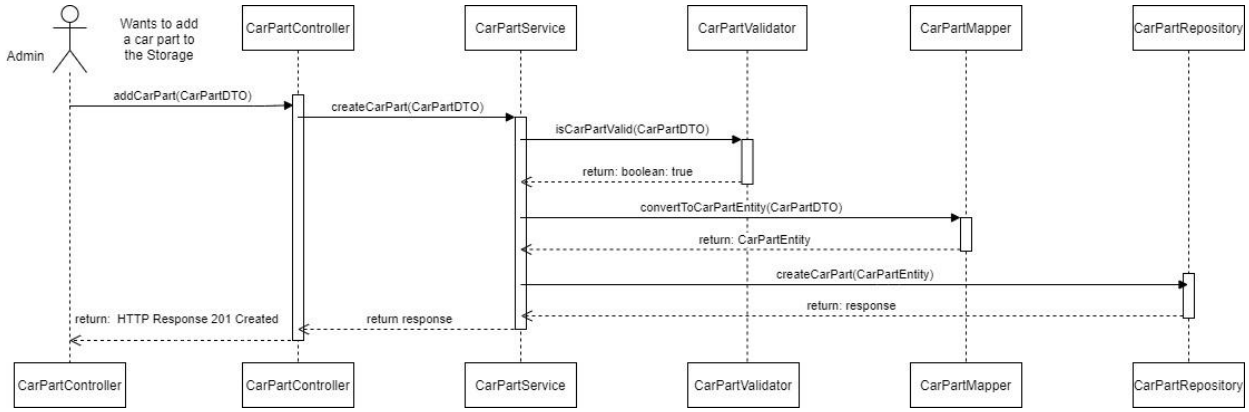
The database diagram presented above presents the main entities from the application and the relationship between them. The attributes of each entity are transposed in the database as separate columns. The main entities would be:

- ✓ ***Admin*** – the administrator of the application, which can administrate the storage products and transactions.
- ✓ ***Customer*** – which is the client that logs in the application, who wants to browse through some products (specifically car parts), add them to their personal cart and order them.
- ✓ ***Car Part*** – models a car part, which has a producer, can be added to the storage of the warehouse and can be purchased by a customer.
- ✓ ***Order*** – once a transaction is made, this is when the order is computed, so that the information about the transaction is stored.
- ✓ ***Cart*** – each customer has a personal cart in which they can add car parts to buy.

3.5 Sequence diagram

The sequence diagram is a diagram representing the behavior of a certain action in the application. It presents the requests and responses and shows how certain components interact in order to fulfill a request.

This specific diagram shows the successful case where an administrator wants to add a new product (car part) to the storage. Therefore, it presents the flow of the data and lifetime of the involved components. It also receives a HTTP response at the end.

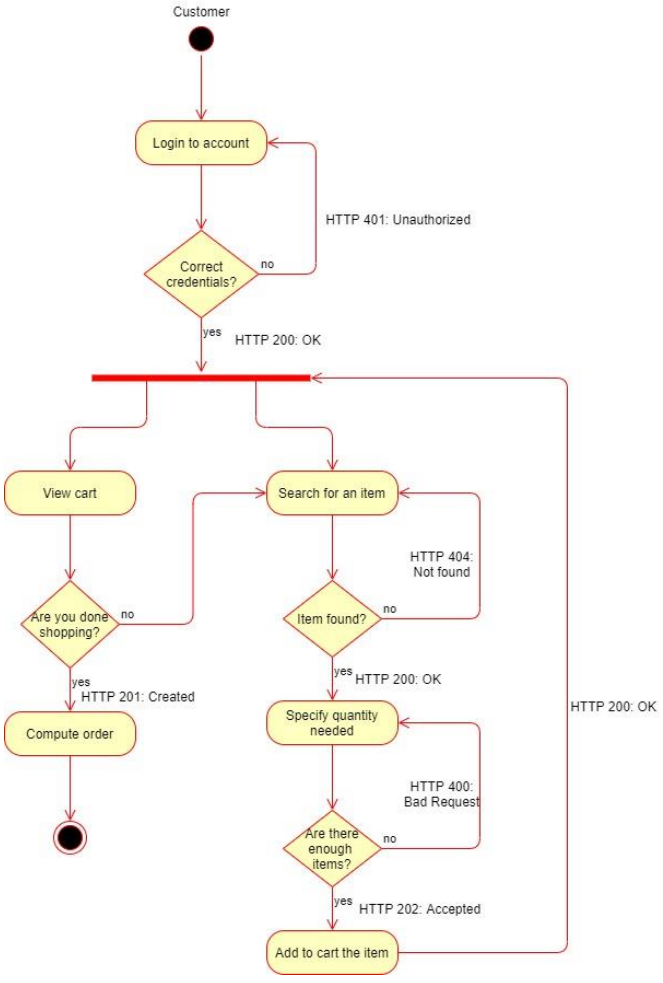


3.6 Activity diagram

The activity diagram describe the behaviour of the application in a certain case. It mostly represents a flow of activities and decisions.

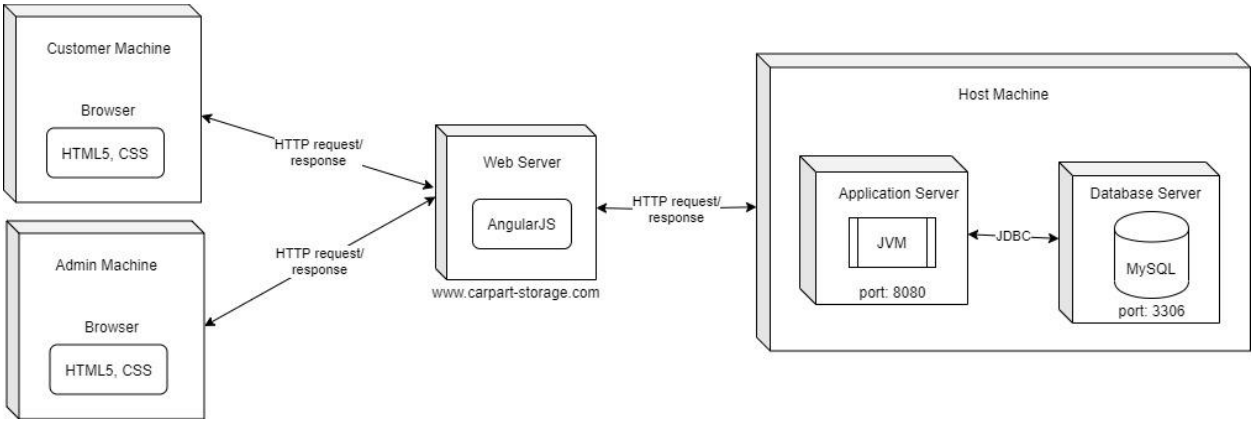
This specific diagram represents the most important activity of the application: a customer searching for a product in the store and buying it. The diagram has 2 complementary flows: adding an item to the cart and then computing an order with that cart.

There are a few activities that the Customer has to go through, presented in the diagram below. Moreover, These requests receive a HTTP response, which is also presented in the diagram:



3.7 Deployment diagram

The deployment diagram resembles the structure of the application, which detemines the fact that it can be used from variuous machines, as it is a web application. The figure below presents how the application and database are created and stored on a host machine and how variuous users (customer or administrators) can access these through a web server.



IV Supplementary specifications

This chapter will be about some functional and non-functional requirements of the application and also some design constraints will be mentioned, which will give a better understanding of how the application is designed.

4.1 Non-functional requirements

The non-functional requirements (NFRs) of the application are mainly represented by the attributes of the application that increase its quality in a specific way, being also appropriate for the scope of the program. This is why they are also called “quality attributes”. Not being focused on the behavior of the application, but on how it can improve the application, the NFRs of the Warehouse application would be:

- ✓ ***Availability*** – the program should be available any time, 24/7, for a client to use it whenever they want.
- ✓ ***Performance*** – the application should have a short response time, so that the requests sent from the clients would get a fast response from the server. This is important for the user, because they should be able to purchase a product as fast as possible, while it is in stock.

- ✓ ***Accessibility across devices*** – the application should be accessible from other types of devices and other operating systems, so it will be easier for the clients to access it.
- ✓ ***Maintainability*** – the maintenance of the application should be as straightforward as possible, so that eventual errors can be corrected fast or other updates can be applied. This is easy as the Layered Architecture allows the developer to access a part very easy.
- ✓ ***Security*** – the information changes regarding the storage can only be performed by an administrator, not by a regular customer. Also, the access to the application will be made by logging in an existent account.
- ✓ ***Post-purchase actions*** – after the customer placed an order, they will be sent an email containing information about the products bought.
- ✓ ***Usability*** – the application should be used easily by the user, having suggestive buttons that can be pressed in order to request something.
- ✓ ***Integrity*** – in order for the data not to be corrupted by any malfunction of the application, when transmitting or storing data. This is why an updated storage is used.
- ✓ ***Backup*** – the products are stored in a storage table in order to have a back-up of the products in case of malware.
- ✓ ***Scalability*** – the application supports multiple clients that make various requests at the same time

4.2 Design constraints

The design constraints are specific to the application and are very important in order to establish the final architecture of the project. Any constraint affects the project, so it is important to know them. For the warehouse application, the design constraints would be:

- ✓ *Language*: **Java** will be used for backend, more explicitly **Java 8** or higher, so that the use of lambda expressions would be possible. For frontend, the project will be made using **HTML**, **CSS** and **TypeScript**.
- ✓ *Framework*: The **Spring** framework will be used for the backend, and **Angular** for frontend.
- ✓ *Database*: The creation of the database will be done using **Hibernate**, and the data will be stored in a relational database, more explicitly **MySQL**. The backend will access data from the database using **JPA Repository**.
- ✓ *Organization*: The project will respect the **MVC** and **Layered Architecture** design.

- ✓ *Access*: The access to the application will be done by logging in an existing account with a username and a password, data that will be stored in the database. The password will not be stored in the database in plain-text, but encoded using **PasswordEncoder**.
- ✓ *Security*: The application will be secured by using **Spring Security**. Validation is also required for the logging in and registering part of the application.
- ✓ *Flow*: As being a Web application, the access from one part of the application to another will be made using **REST APIs**.

V Testing

< Se va discuta la laborator./>

5.1 Testing methods/frameworks

5.2 Future improvements

VI Bibliography