HB Products Mobile Application

Version history

Description	Version	Init	Revised	Approved	Date
Draft of document	0.1.0	HS, KR, MD	POV	HS, KR, MD	05-09-2019
Small corrections	0.1.1	HS, KR, MD	POV	HS, KR, MD	18-11-2019
Final Version	1.0.0	HS, KR, MD	HS, KR, MD	HS, KR, MD	19-12-2019

Hristo Stoyanov(253911)

Mariyan Deligalabov(253896)

Konstantin Ralev(253640)

Supervisor: Poul Væggemose

VIA University College

VIA University College

106.025 **Characters with Spaces**

ICT Engineering

7th semester

19 December 2019

Table of content

Abstract

The purpose of this project was to develop an application that would improve HB Products marketing and customer service. This was done by developing an application which had the following features implemented. A QR code scanner which was built for the customers ease of distinguishing a product. A product can be identified by either scanning it or selecting it from a product catalog. A list of features is then displayed to the customer. Some of them are a 3D model of the selected/scanned sensor, User Manual and a Quick Start Guide. When a sensor is selected the customer can also make a sensor enquiry or visit the selected sensor on HB Products' web shop. One of the application requirements was that it needs to be developed for iOS and Android mobile operating systems. The choice of a development platform was Xamarin because there is a single code base, which can be compiled on both platforms. All the application servers and its database were deployed and hosted on Microsoft Azure.

Overall this project was a success. The project passed its acceptance testing and all of its required functionalities were implemented properly and accepted by the stakeholder. This project report documents the development of this project separating it into analysis, design, implementation and testing sections. A supplementary document to this report is the process report which focuses more on the process of developing the application.

1 Introduction

In today’s technology market every company wants to be one step ahead of its competitors. One way to stay on top is to add services that benefit the customers and makes them understand why your products are easier to use and maintain than the others available on the market.

These days mobile applications are the new trend. Every company wants to have one, mostly for marketing and to increase their sales. According to statistics, more users buy stuff online using mobile apps than using websites (Target Informatics Pvt. Ltd., 2018) because smartphones are accessible by all and they can connect to the web no matter the location.

HB Products is a development-oriented company, which specializes in the development and production of sensors for industrial refrigeration systems. (HB Products A/S, 2019)The company is currently growing and expanding its facilities. Therefore, HB Products would like to have their own cross-platform mobile application which will improve their customers’ experience and increase their market share.

The application will be developed using Xamarin. Some of the features that are going to be implemented are 3D model viewer which improves the perception of how the sensor looks, QR code scanner which will make identifying the type of the sensor easier, FAQ(Frequently Asked Questions) section, Contact functionality, and other small features.

The project is only going to be proof of concept due to time restrictions. The delivered product will be a prototype that will be used for demonstrative purposes. For full project description please check Appendix L.

The following sections will explain how each of the project phases was carried out and give interesting examples.

- Analysis – The analysis section explains the requirements that were accepted by the product owner and the diagrams that follows the requirements (Use case, domain model, state machine, system sequence, activity)
- Design - The design section expands on the analysis and explains the system design, choice of technologies, choices of models and methods, system architecture and prototyping. It is more technically oriented than the analysis phase.
- Implementation – This section shows examples of how the design was implemented by showing code snippets from important parts of the system.
- Testing – In this section, the implemented code gets tested to verify that it meets the expectations of the product owner and/or the development team.

2 Analysis

This section of the project report gives detailed information about the analysis phase of the project and what methods were used during the period. The main topics that the analysis phase elaborates on are the establishment of requirements, problem domain and diagrams that will be later on used in the design phase. The requirements in this project are prioritized by using the “MoSCoW” prioritization technique.

2.1 Requirements

MoSCoW prioritization

The technique used for prioritizing the requirements for this project is “MoSCoW”. This method has 4 levels of priority which describe how much each requirement will benefit the final product. In the requirements table below acronyms are used to describe the priority of each requirement – see table listed below.

Acronym	Definition
M	Must have
S	Should have
C	Could have
W	Would not have

Must have – These provide the Minimum Usable Subset (MUS) of requirements which the project guarantees to deliver.

Should have - Important but not vital

Could have - Wanted or desirable but less important

Would not have - These are requirements which the project team has agreed it will not deliver. They are recorded in the Prioritised Requirements List where they help clarify the scope of the project and to avoid being reintroduced ‘via the back door’ at a later date. (Volkerdon, 2018)

2.1.1 Functional Requirements

ID	Requirement	Priority
1	The customer should be able to save his/her contact credentials	S
2	The application should autofill contact forms using the saved customer credentials.	S
3	The application should have a sliding menu drawer for navigating.	M
4	The application should have a QR scanner.	M
5	The application should be able to recognize the model of a sensor by scanning its QR code.	M
6	The application should be able to provide the customer with the user manual for a selected product.	M
7	The application should have an online HB Products webshop.	M
8	The application should be able to present the customer with a 3D model of the selected sensor.	S
9	The application should have a Contact us tab.	S
10	The application should have frequently asked questions tab.	S
11	The Contact us tab should have “Write an email” and “Chat with us” options.	S
12	The customer should be able to mark products as favorites.	C
13	The customer should be able to interact with the 3D model using touch inputs	S
14	The product owner should be able to send push notifications to the users of the app.	C
15	The application should have a chat functionality used for customer support.	C
16	The customer should be able to save the chat session for reviewing later.	C
17	The customer should be able to make an enquiry for a specific product.	C
18	The application should prompt the customer for permissions when they are required.	S

2.1.2 Non-Functional Requirements

ID	Requirement	Priority
1	The application should run on Android and iOS mobile operating systems.	M
2	The application should be developed in Xamarin using C#.	M
3	The application should have a sliding menu drawer for navigating.	M
4	The application should have a QR scanner.	M
5	The application should be able to recognize the model of a sensor by scanning its QR code.	M
6	The sensor data should be saved in a database	M
7	The application should be able to provide the user with a user manual for a selected product.	M
8	The application should let the user know whenever the application is processing information (show loading icon).	M

2.2 Problem Domain

The manufacturing industry is very important for the Nordic Sea Region economy and remains a driver for growth. That is why as a part of the GrowIn 4.0 project HB Products wanted to improve their sales and customer service with the implementation of some newer technology. (Interreg - North Sea Region Programme, 2014-2020)

Currently one of the problems of the company is that their sensors look very similar to each other, this is the reason why they wanted to help the customer identify them in an intuitive way. Furthermore, most of the sold sensors are already mounted and some of the customers don't know how the sensor looks so they also wanted a 3D representation of it. Another problem that they have is that currently, they do not have a Customer Support chat and an FAQ section on their website that's why the company wanted these features also implemented.
Our project was designed around the company's requirements in a way that we think that it can help HB Products expand their market, increase their sales and improve their customer service.

2.3 Domain Model Diagram

The development team constructed the domain model, which demonstrates how the objects relate to each other. The objects presented in the diagram are determined from the requirements.

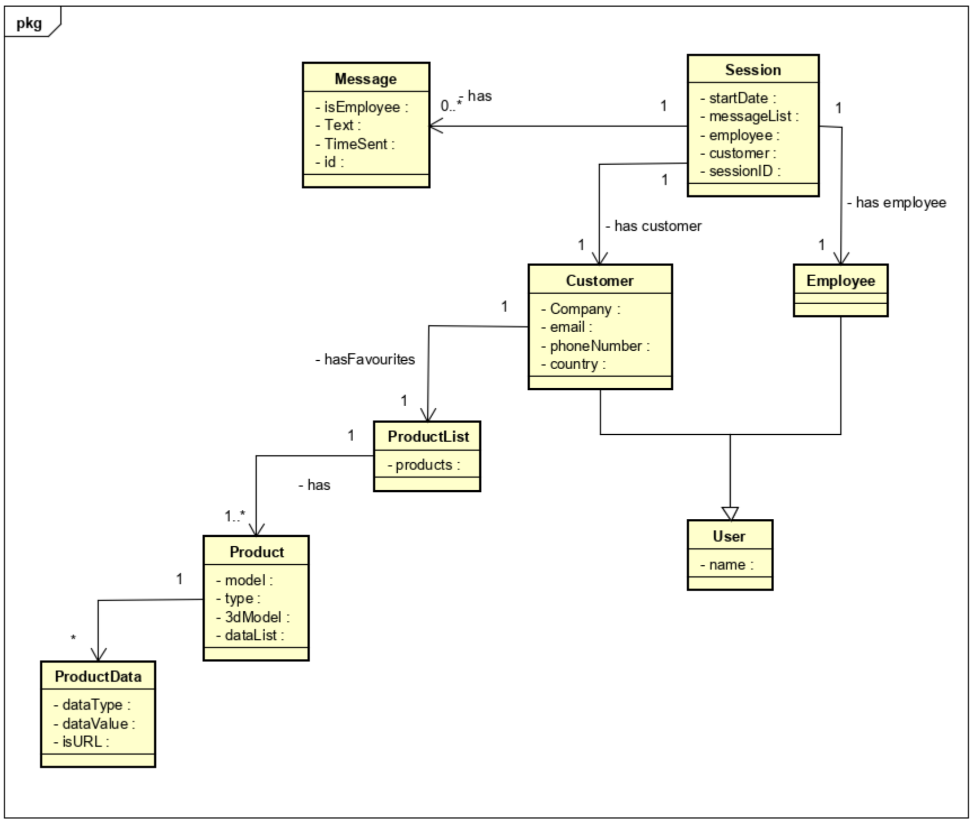


Figure 2.1: Domain Model Diagram

It can be seen that the Session contains a list of messages, startDate which is used as a timestamp of when the session started and two users – a customer and employee. Customer, on the other hand, is extending the User class by adding a few more fields that are unique to the customer.

The Product class is containing information about a single product. The data about the product like user manuals, pictures, quick start guides, etc. are stored in a list of ProductData objects inside of the Product class. The ProductData class also has a Boolean ‘isURL’ which provides information if the data contains a link.

ProductList is a class that is added to enable scalability and extensibility in the future – it will enable you to add filters and send product lists to the web shop to make orders.

2.4 Use Case Diagram

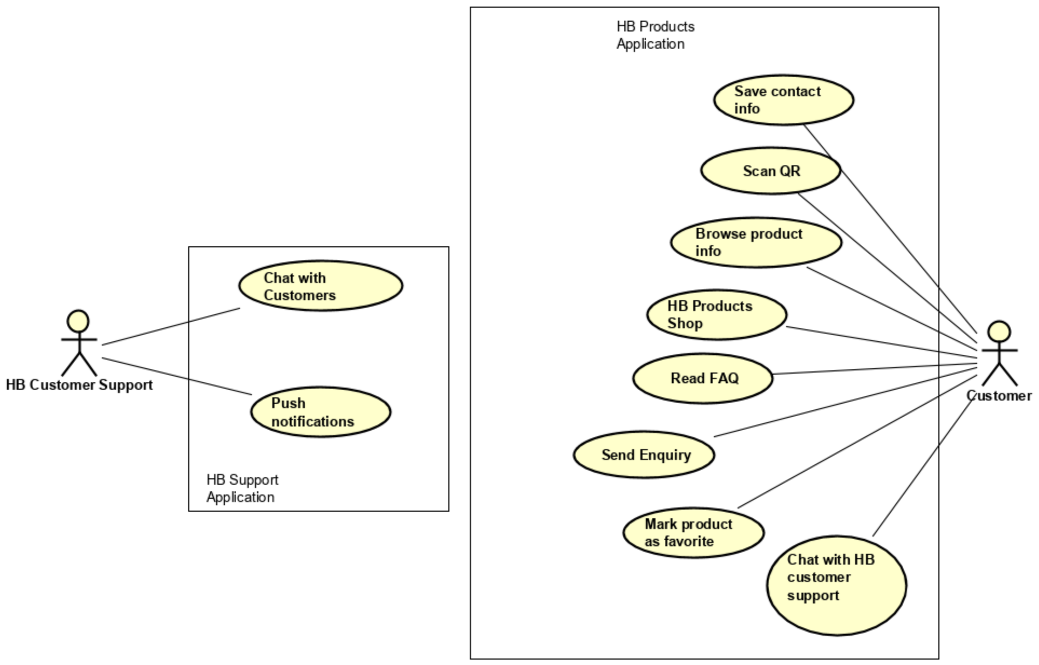


Figure 2.2 Use Case Diagram

Based on the requirements, which were created at the beginning of the Analysis Phase, a Use Case Diagram was drawn to visualize the two actors that will use the system and give a better overview of the prototype. On the Figure 2.2 above can be seen what functionalities both actors will have.

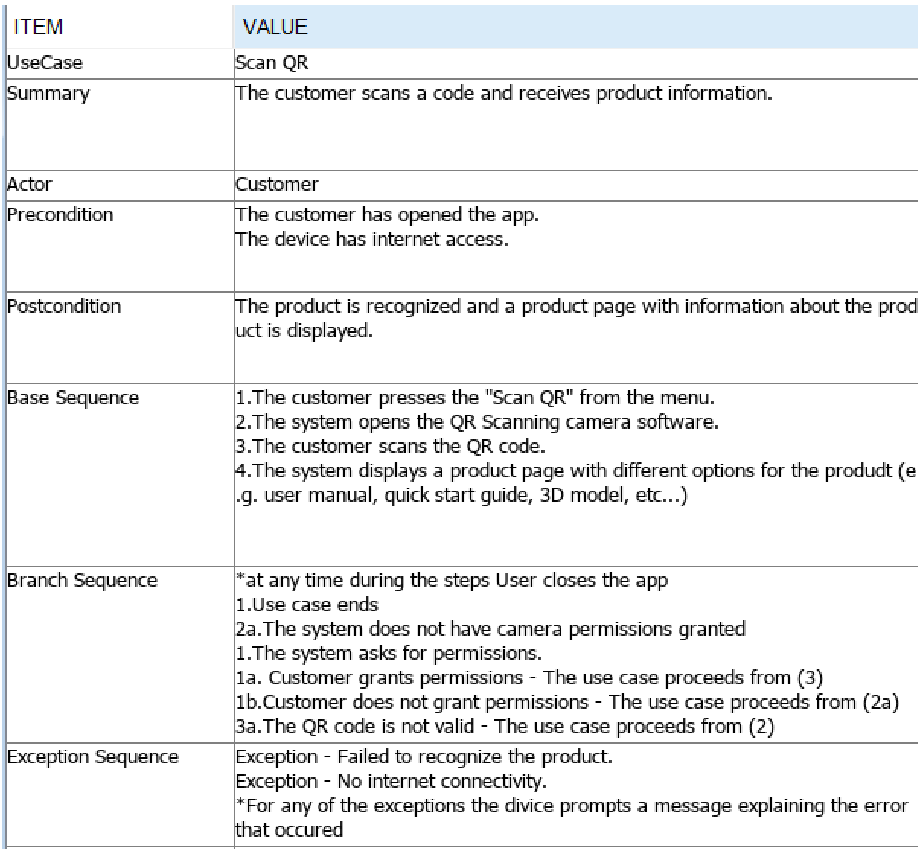
The two actors will have separate mobile applications. Each mobile application will have the features, which are given to the corresponding actor. For better quality Use case diagram please check Appendix K.

2.5 Use Case Descriptions

The use case diagram contains multiple use cases. Each use case is thoroughly described by its use case description. In this section, only a limited amount of use case descriptions will be shown. The rest of the use case descriptions could be found in Appendix D. The use case descriptions are then explained by the Activity diagrams which give a better understanding of the flow of actions. For more information please check Section 2.7.

2.5.1 Scan a QR

The following use case description describes how the user gets product information by scanning the QR code on a product.

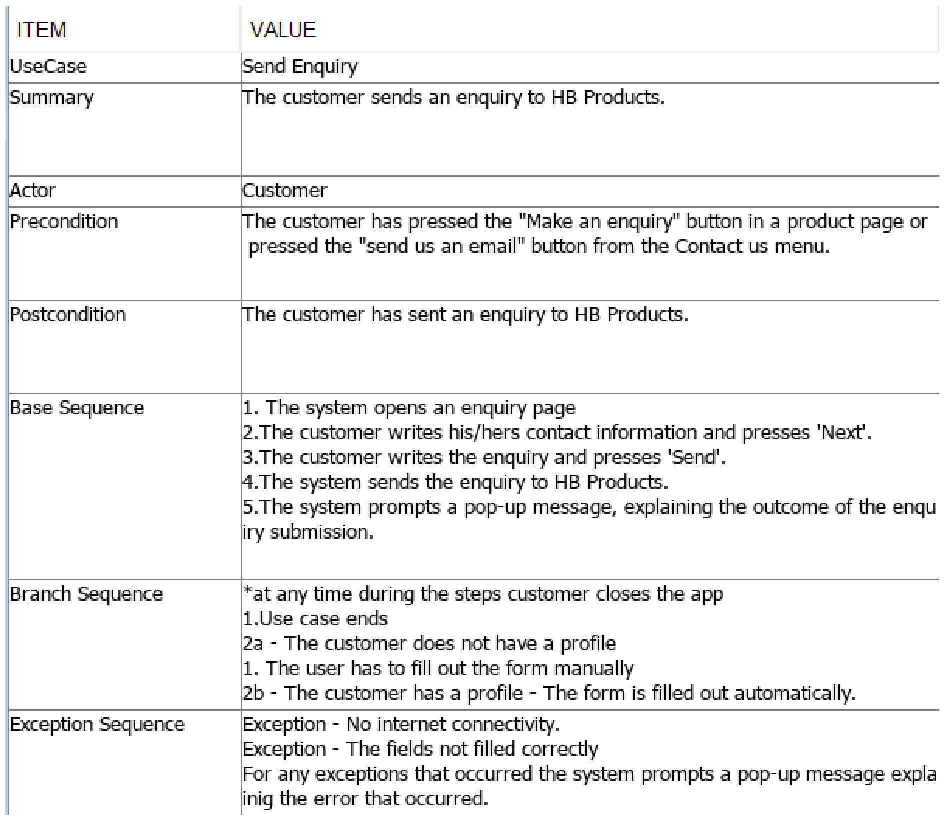


ITEM	VALUE
UseCase	Scan QR
Summary	The customer scans a code and receives product information.
Actor	Customer
Precondition	The customer has opened the app. The device has internet access.
Postcondition	The product is recognized and a product page with information about the prod uct is displayed.
Base Sequence	1.The customer presses the "Scan QR" from the menu. 2.The system opens the QR Scanning camera software. 3.The customer scans the QR code. 4.The system displays a product page with different options for the produdt (e .g. user manual, quick start guide, 3D model, etc...)
Branch Sequence	*at any time during the steps User closes the app 1.Use case ends 2a.The system does not have camera permissions granted 1.The system asks for permissions. 1a. Customer grants permissions - The use case proceeds from (3) 1b.Customer does not grant permissions - The use case proceeds from (2a) 3a.The QR code is not valid - The use case proceeds from (2)
Exception Sequence	Exception - Failed to recognize the product. Exception - No internet connectivity. *For any of the exceptions the divice prompts a message explaining the error that occured

Figure 2.3 - Scan QR Use case description

2.5.2 Send Enquiry

The following use case describes how a user creates and sends an enquiry to HB Products.



ITEM	VALUE
UseCase	Send Enquiry
Summary	The customer sends an enquiry to HB Products.
Actor	Customer
Precondition	The customer has pressed the "Make an enquiry" button in a product page or pressed the "send us an email" button from the Contact us menu.
Postcondition	The customer has sent an enquiry to HB Products.
Base Sequence	1. The system opens an enquiry page 2.The customer writes his/hers contact information and presses 'Next'. 3.The customer writes the enquiry and presses 'Send'. 4.The system sends the enquiry to HB Products. 5.The system prompts a pop-up message, explaining the outcome of the enqu iry submission.
Branch Sequence	*at any time during the steps customer closes the app 1.Use case ends 2a - The customer does not have a profile 1. The user has to fill out the form manually 2b - The customer has a profile - The form is filled out automatically.
Exception Sequence	Exception - No internet connectivity. Exception - The fields not filled correctly For any exceptions that occurred the system prompts a pop-up message expla inig the error that occurred.

Figure 2.4 - Enquiry Use Case Description

2.6 System Sequence Diagram

To improve the understanding of how the customer and system communicate between each other system sequence diagrams are made for each use case. This section will not contain all the system sequence diagrams but only a selected number. The rest of the system sequence diagrams could be found in Appendix E.

The following diagram figure 2.5 displays how a customer can browse product data about the currently available products.

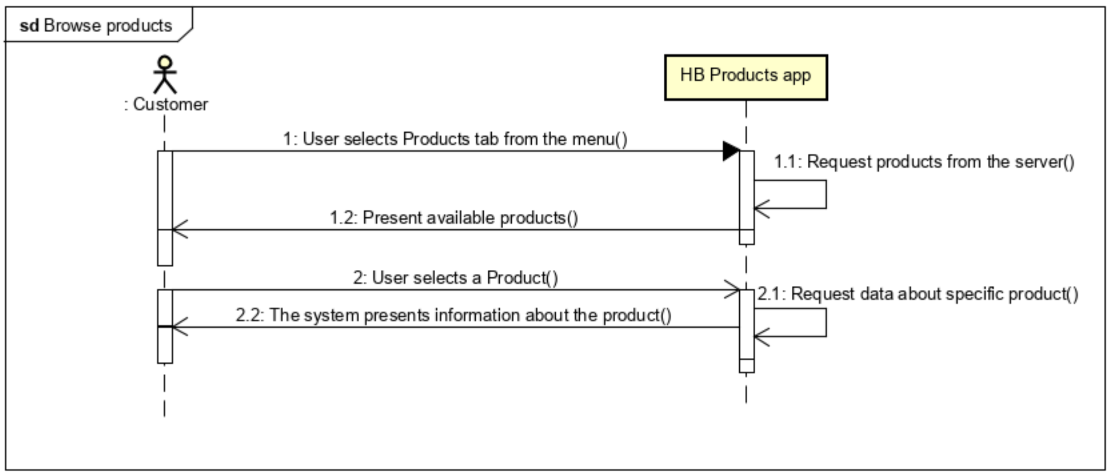

sd Browse products
: Customer
HB Products app
1: User selects Products tab from the menu()
1.1: Request products from the server()
1.2: Present available products()
2: User selects a Product()
2.1: Request data about specific product()
2.2: The system presents information about the product()

Figure 2.5 – Browse Products System Sequence Diagram

2.7 Activity Diagrams

The size of the activity diagrams does not fit on A4 page. Thus, they will not be mentioned in this report. The activity diagrams could be accessed in Appendix J.

3 Design

This section will describe the architecture, choice of technology, models and methods, ER diagram, class diagram, sequence diagram, deployment diagram and design patterns used in the project. In order to achieve good, future proof and scalable design, architectural patterns are used in the project. The goal of the system design is to design a system in a way that lowers the dependencies of each object making it easy to modify and upgrade.

3.1 Design Approach

3.1.1 Interaction Design

In the 6th semester, all team members chose an elective course in Interaction Design. After learning more about how this way of designing a product works and its benefits, interactive way of designing was chosen as the design technique for this project.
The Interaction Design cycle starts with "Establishing the Requirements" which was done in the Analysis phase. After finishing the Analysis phase and getting approval of the specified requirements the project continued to the next Interaction design stage, which was creating several project designs based on the requirements and deciding, which one is the best for the project.

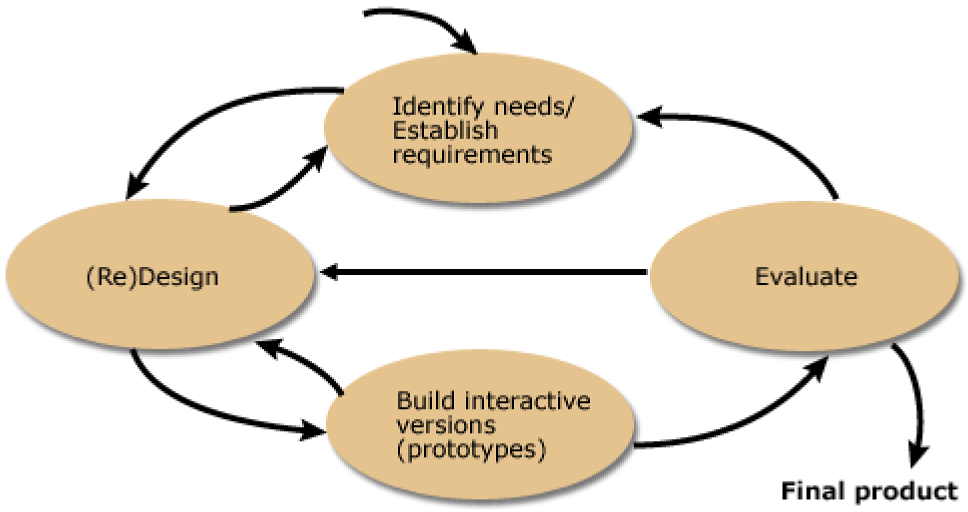

Identify needs/ Establish requirements
(Re)Design
Evaluate
Build interactive versions (prototypes)
Final product

Figure 3.1 – Interaction Design Flow (MasterGruppeG, 2012)

3.1.2 Design Alternatives

When designing the project the usability, usefulness and intuitive design were the main focus. The initial design (Figure 3.2) of the system contained a mobile application, which had a QR code scanner, 3D model viewer and a list of products. Each of the products, when chosen, will present the user with its specs and a manual. The product specification and the manuals are stored in a database.

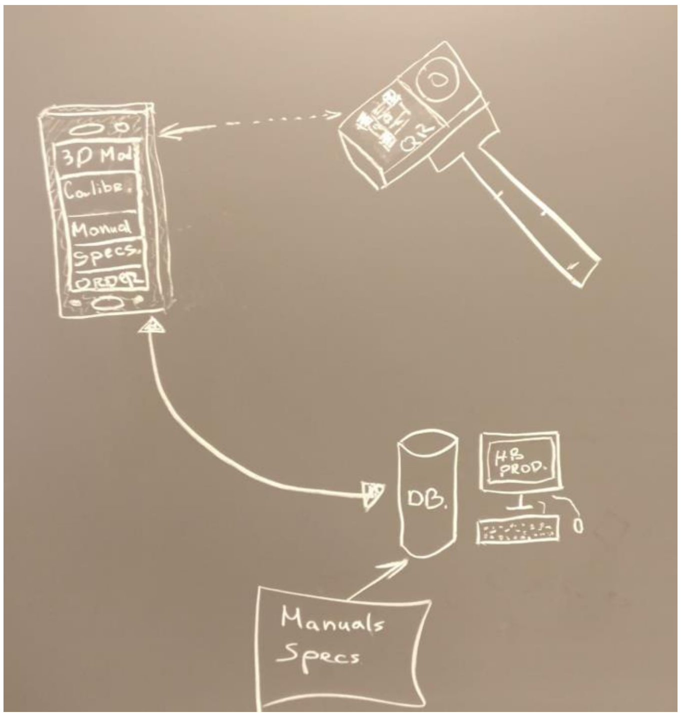


Figure 3.2 Initial System Design

After this design of the project was shown and discussed with the product owner, it was decided that it is simply not enough for a bachelor project and it needs to be changed. Some of the requirements were not taken into consideration and some parts of the requirements were not clear enough at this point in the project. As an outcome of this discussion a final design (Figure 3.3) of the project was created and approved by the product owner.

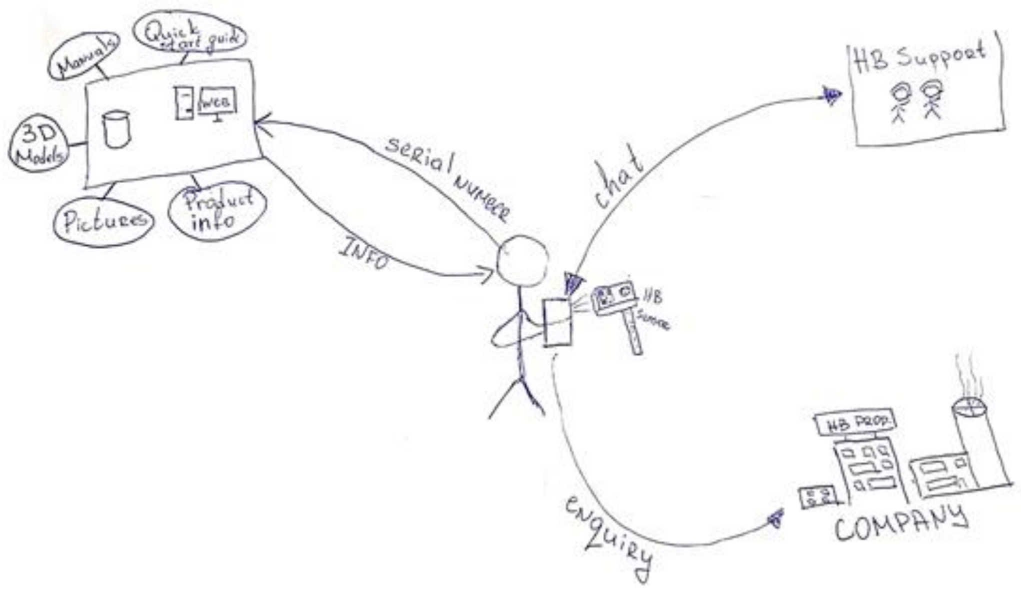


Figure 3.3 Final System Design

3.2 Prototyping and Choice of Technology

Having a final design and requirements the prototyping started. The group needed to do a lot of research at this stage to be able to decide on the technologies that need to be used. The mobile application had to be cross-platform (iOS/Android). After researching the different options that were available like creating two separate applications on native development platforms, Apache Cordova, Ionic, Flutter and Xamarin. It was decided to go with Xamarin. More about why Xamarin was chosen and what were the other alternatives can be found in section 3.5.1.

The next step was to start researching how to implement a 3D representation of a sensor into the app. This was one of the most challenging tasks. Several prototypes were developed using different technologies like Urho3D, OpenGL and Unity. UrhoSharp was the final choice with supposed support for Xamarin apps and the correct 3d model format support. The prototype of this engine was deployed to an

Android and an Apple device and it worked flawlessly. More about why UrhoSharp was chosen and what were the other alternatives can be found in section 3.5.3.

Developing a system without a design pattern managing the view and model classes is simply a bad system design. There are more than a few design patterns that were invented for the purpose of making a system that could be easily understood by other programmers after a short inspection of the packages/subsystems. The team had previous experience with the Model-View-Controller (MVC) design pattern but having in mind that we have the possibility to use two-way data binding that C# supports, MVC was not a candidate that would fit our needs as well as MVVM.

The Model-View-ViewModel was a design pattern that none of the team members have worked with in the past so making a prototype was a logical step in the project development. The MVVM design pattern is also recommended to use when using data binding by the producer of Xamarin – Microsoft. (Microsoft, Architecture - Xamarin, 2017) It was decided to make a prototype application including some of the basic functionalities in order to gain more knowledge about how the system would work. The team decided to implement the sliding menu, used for navigating throughout the application using the MVVM Design pattern. The menu was populated with some of the pages that would be used later in the final version of the app. The list of pages in the menu was not hardcoded into the menu itself but a dynamic listview was created which would expand every time you add a menu option inside of the menu. The list of pages in the menu was data bounded to a list of menu items created in a separate class.

Then after having a working prototype of the different technologies the team had to start combining all of the modules into one application in a way that was scalable; each module was reusable and maintainable. At this point in the design phase, a working prototype of some of the features was issued and shown to the company for evaluation purposes.

3.3 Evaluation

The finished design was shown to the COO of the company and it was approved. Based on the approved design a prototype was created of the application. The COO wanted to present the project to a board meeting of the company and wanted some “fancy buttons” to be added for the demonstration purposes - this was done. The required “fancy buttons” were with an image background to make the UI “fancy”. A video of the prototype application in use and some screenshots was sent to the COO of the company. The outcome of this evaluation was positive, and the group continued with the implementation.

3.4 Architecture

For the development of the project, 3 tier architecture was chosen because the development team had previous experience with this type of system architecture. This architecture provides many benefits for the production and development environment by modularizing the user interface, business logic and data storage. (Jinfonet Software, 2019)

3.4.1 Presentation tier

The presentation tier is the tier containing all the user interfaces and its supporting classes. This layer could be also explained as the one that the user directly interacts with. The design of this part of the project was carried out by following software architecture rules as Model-View-ViewModel (MVVM) design pattern, SOLID principles, Neighbor Communication Principles (NCP). Adding to that the Class Naming Principle (CNP) was partially followed. More information about the patterns could be found in section 3.6 Design patterns and diagrams.

3.4.2 Business Logic Tier

The business logic tier is the tier where the application business logic is contained. From here all core application capabilities are driven and this is the gateway between the Presentation tire and the Data tire. Examples of technologies for developing this tier is: Java, Python .NET C#, C++, etc.

3.4.3 Data Tier

The data tier is the layer where all the data is stored and processed. Examples of data tiers systems are Oracle, PostgreSQL, MySQL, MongoDB, etc. Data is accessed by the application layer using an API request to the business layer, which on its side is connected to the data layer and has access to the data and the database functions.

3.5 Technologies

3.5.1 Cross-platform mobile app development – Xamarin

The technology for developing the cross-platform application used in this project is called “Xamarin”. Xamarin is a product owned by Microsoft and is used for developing cross-platform mobile applications that can run on the Windows mobile platforms, Android and iOS. Xamarin extends the .NET platform with tools and libraries specifically for building apps on iOS, Android, macOS, and more. (Microsoft - What is Xamarin?, 2019)
Xamarin gives a possibility to the developers to write the logic and UI once and then build on all mobile platforms successfully – see figure 3.4

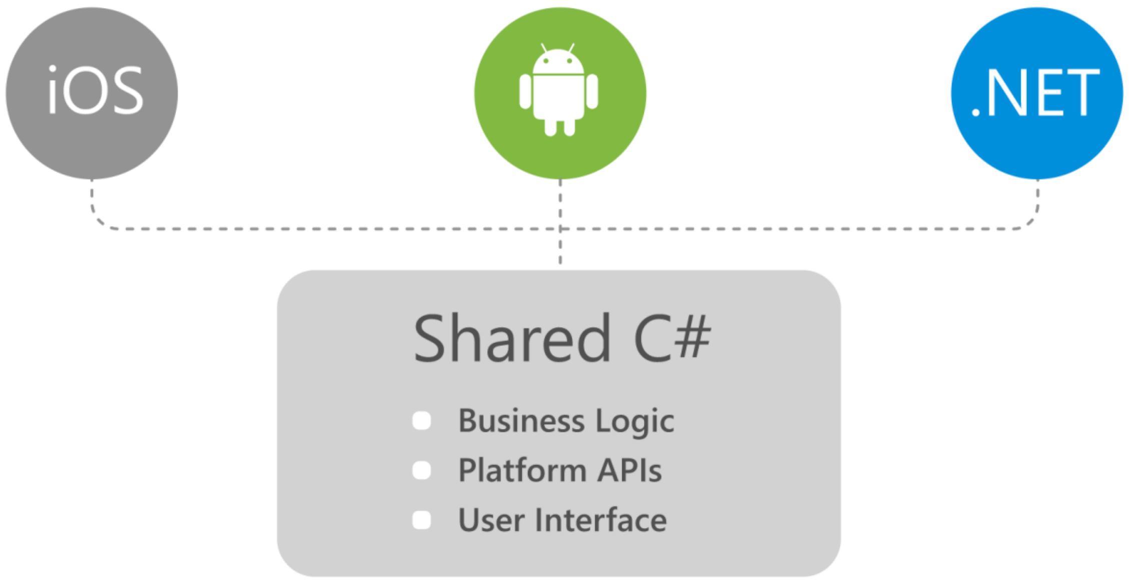


Figure 3.4 – Xamarin Structure

The code-behind is written in C# and the user interface is written in Extensible Application Markup Language (XAML) – a declarative XML-based language developed by Microsoft. During compilation, the compiler provides the device with a native or integrated .NET application and runtime. The compiler also performs code optimization (e.g. removes unused code and classes to reduce the application size).

Alternative technologies to Xamarin

There are many possibilities for cross-platform development at this point in time such as Flutter, React Native, Ionic, Unity, Apache Cordova, etc.

All of those platforms provide superior functionalities, but the main selling point for choosing Xamarin for this project was that the logic is written in C#. The development team is more familiar with C# than with any of the other languages used in the other platforms – mainly JavaScript or JavaScript-based languages. One of the other candidates for a cross-platform development platform that uses C# was Unity. Unity was not chosen because it is mostly preferred for game development – it does not support native mobile UI elements, something that Xamarin does. The native UI elements are intuitive to the users and greatly improves the user experience. Therefore, Xamarin was chosen over Unity and the other candidates.

3.5.2 QR Code Scanner – ZXing

One of the application's main features is the scanning of the sensor's QR code. That is why it was important to research and decide on how this is going to be implemented. Since the development environment was already chosen the development team started researching QR code scanning options that support Xamarin. While researching ZXing (Zebra crossing) and Scandit were found. The technology of choice was ZXing simply because it was free, cross-platform compatible, with good support and documentation. ZXing is a NuGet Package that is added to the Visual Studio project and with a few modifications it can run on both mobile platforms – Android and iOS.

3.5.3 3D Rendering Engine – Urho Sharp

The choice of the 3D Rendering Engine was one of the hardest ones. Urho Sharp was the engine of choice. UrhoSharp is a powerful 3D Game Engine for Xamarin and .NET developers. It is similar in spirit to Apple's SceneKit and SpriteKit and includes physics, navigation, networking and much more while still being cross-platform.
It is a .NET binding to the Urho3D engine and allows developers to write cross-platform code that can target Android, iOS, Windows, and Mac with the same codebase. (Dunn, 2017)

Alternative technologies to Urho Sharp
There are many possibilities for mobile 3D Engines currently such as Unity and OpenGL.

All of those technologies were considered in the Design state of this project and prototypes were developed. Unfortunately, Xamarin could not use Unity as a library because it is currently under development. OpenGL was also considered but was not the final rendering option because of limited 3D object file support and performance on mobile. The final choice was Urho Sharp because of its Xamarin support and its superior performance over the alternatives.

3.5.4 Database System – PostgreSQL

The databases that were considered during the design phase were Oracle Database, PostgreSQL, MySQL, MongoDB, Firebase, etc. The database of choice was PostgreSQL. It was chosen for this project because the team had previous experience with this database and felt more confident to work with something familiar. PostgreSQL is a powerful, open-source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance. (The PostgreSQL Global Development Group - About, 2019)

3.5.5 Hosting – Microsoft Azure

At this point in the project, the team figured out that a hosting service will be needed because it was not a good idea to host everything locally due to limitations. A number of hosting services like Microsoft Azure, Google Cloud and AWS were considered. After researching what all of them offer the group decided to go with Microsoft Azure. Microsoft Azure is a cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers. It supports many different programming languages, tools, and frameworks. (Wikipedia - Microsoft Azure, 2019)

Based on the predefined requirements the group knew that a Webservice, PostgreSQL Database, Notification Service and a Cloud-Based Email delivery platform will be needed. Microsoft Azure offered all of those features in one place and a free trial model which was long enough for the purpose of the project.

3.5.6 Server – WebAPI

Security was also something that the group was concerned about. Giving users direct access to the database is not a good idea. After going through the alternatives of making the connection between the database and application, the team chooses to go with ASP.NET Web API. (Wikipedia - Web API, 2019)

A server-side web API is a programmatic interface consisting of one or more publicly exposed endpoints to a defined request-response message system, typically expressed in JSON or XML, which is exposed via the web—most commonly by means of an HTTP-based web server. (Wikipedia - Web API, 2019)

This choice was made because both Xamarin and the .NET Web API are developed in C# and Microsoft Azure supports server hosting for this Web API projects.

3.5.7 JSON

The team needed a way to send information between the server and the application. Two ways were considered. One was to receive a byte array which after that should be put into a buffer and then parsed into an object. The other way was to use JSON which can transform any object into a serializable string that could be added as HTTP request content in the request body. The team decided to go with the JSON approach because of its simplicity and some of the team members have used it in previous projects, so they were familiar with it. (Wikipedia - JSON, 2019)

JSON is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value). It is a very common data format, with a diverse range of applications.
(Wikipedia - JSON, 2019)

3.5.8 Notification Service – Azure Notification Hub

One of the company’s requirements was “The product owner should be able to send push notifications to the users of the app.”. The group needed a way to handle notifications on both platforms (iOS and Android). Both of the platforms use two completely different ways of handling their notifications. For example, Apple uses APNS (Apple Push Notification service) and Google uses FCM (Firebase Cloud Messaging). The services taken into consideration were OneSignal, Firebase Cloud Messaging, and Azure Notification Hub. Azure Notification Hub was chosen because Azure was already the choice of a hosting platform, so its notification hub looked like the way to go as it also supported both APNS and FCM.

3.5.9 Cloud-Based Email Delivery Platform – SendGrid

The application had to have a “Make an Enquiry” and “Contact us” functions which were supposed to send emails to the company. At one of the meetings with the company, the group talked with the product owner about this and it was decided that clicking the button and transferring you to the default Email app is not the perfect solution. That’s where the idea about a Cloud-Based Email Delivery was born. When researching the alternative platforms one of the most important features that it needed to have was to be free, easy to implement and if possible be hosted on Azure. SendGrid was found. It fitted perfectly into the project because it had a free plan with a certain amount of emails per month and it supported Azure.

3.6 Design patterns and diagrams

3.6.1 Design patterns

Model-View-ViewModel(MVVM)

The Model-View-ViewModel design pattern’s role in software development is to separate the business and presentation layers of the application.

To ensure the easy scalability and testability in a software product, the design should be done in a way that eliminates tight coupling between the elements in the system. In a tightly coupled system a simple UI change could lead to rewriting of the whole code-behind.

The three main components in the MVVM pattern are: model, view and view model. Each of the components has its own purpose. (Dunn, et al., 2017) To obtain a better understanding of the relationship between the components check out figure 3.5

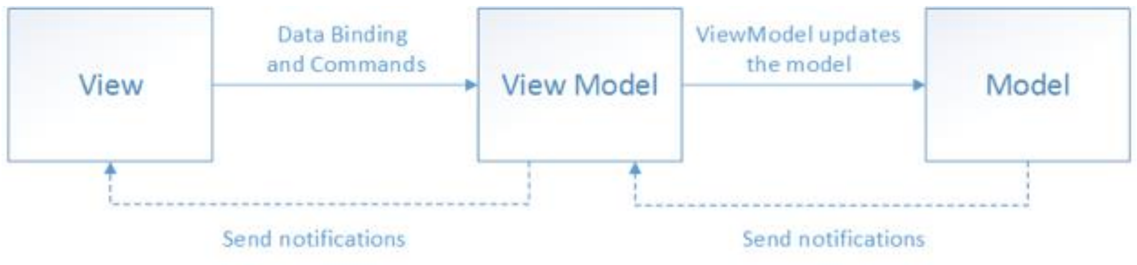


Figure 3.5 – MVVM Pattern Structure

As shown in the diagram above the View is not connected to the model and does not use it to update the UI elements that the user can see and interact with. Also, the model does not know that it is observed and does not have instances of the view model. The communication between the three layers is done as follows:

- **View**

  The view is binding to properties and calling commands in the view model. The commands are just like methods but do not provide anything in return. The commands are usually bound to interactable components in the UI and perform a certain action inside of the view model. Commands are usually the way to go when it comes to MVVM design pattern, but it is not suitable in all cases (e.g. selecting an item from a list – the view model cannot know exactly which item is chosen). In cases like that the view has to handle the click and call a method in the view model.

  Data bindings are used for two-way communication between the UI and the view model. Any change in the bound object will be represented in the UI automatically without any updates in the code.

- **View Model**
  The view model has properties of the model and can modify them but does not “know” anything about the view. Instead, the view model has a notification interface that is used to notify the view. The view model is also responsible for the logic behind the view. The enabling and disabling of UI elements is also done by binding them onto properties of the view-model (e.g. when an operation is loading). The view model is also responsible for coordinating the view to the model properties. In some cases, the classes of the model could be designed so that the view can directly bind to some of the model’s properties. In that case, the model classes should be designed to support data binding and raise “OnPropertyChanged” events.
  Using the view model class is also useful in cases of hardware-intensive operations – it keeps the load off the UI thread. Doing operations on the UI thread frequently freezes the UI and results in unpleasant user experience.

- **Model**
  The model classes are the ones encapsulating the data of the application. Those classes are extracted from the domain model of the system. The app’s users never get to see or interact with this part of the system.

3.6.2 Class diagram

In order to keep the system organized the subsystems were separated in a number of packages:

Model – Contains the model classes of the application - see Figure 3.6

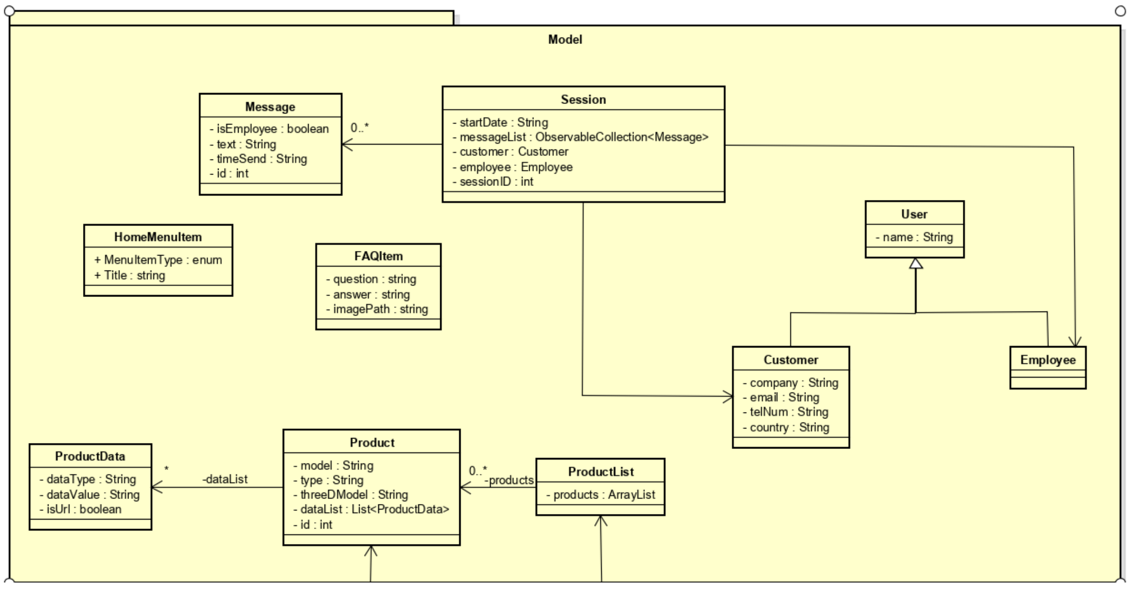


Figure 3.6 – Model Package of the Class Diagram

- **Services** – Contains the managers that connect to the web API and other services used inside of the application – see Figure 3.7

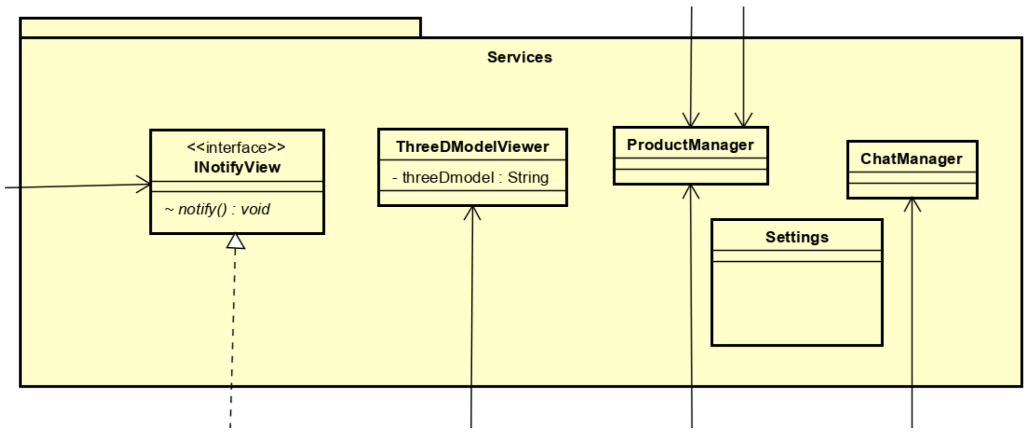


Figure 3.7 – Services Package of the Class Diagram

View – Contains the view classes of the application. This is the part of the system that the user interacts with – see Figure 3.8

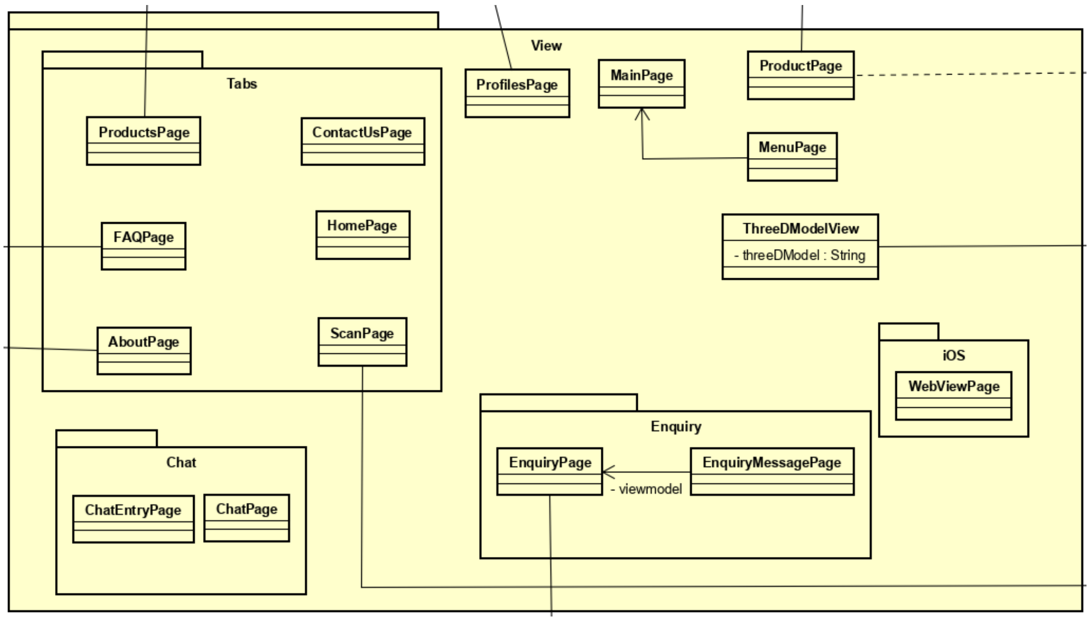


Figure 3.8 – View Package of the Class Diagram

View Model – Contains the view model classes that contain instances of the model and is observed by the views – see Figure 3.9

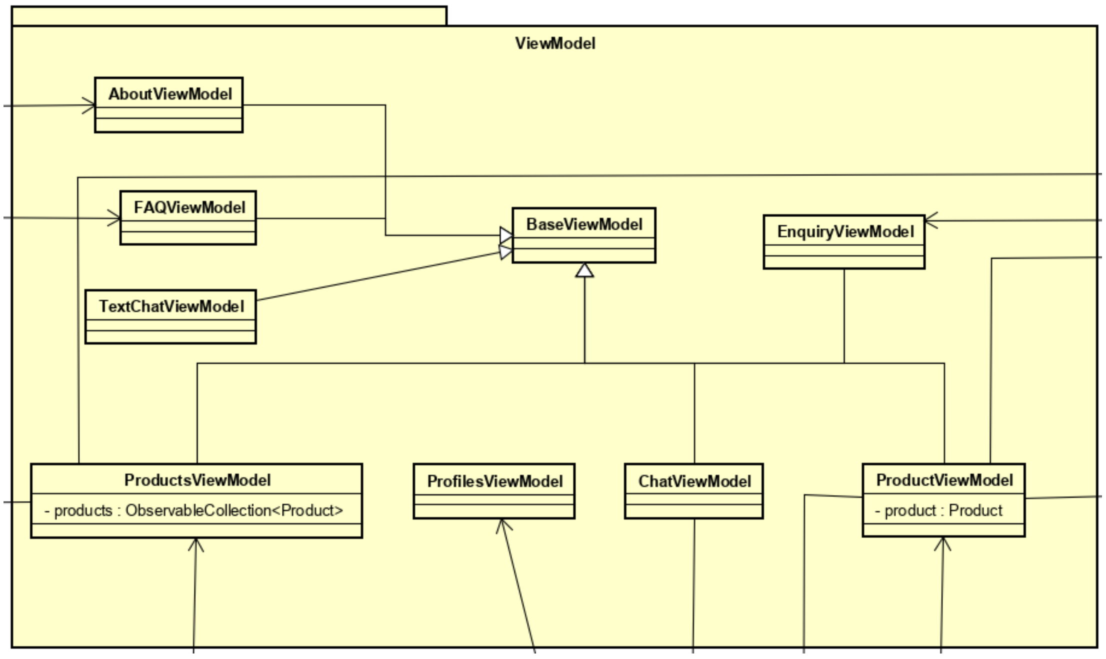


Figure 3.9 – View Model Package of the Class Diagram

- To see all class diagrams including structural and complete class diagrams please check Appendix F.

3.6.3 Sequence Diagram

The sequence diagrams for this project are made with the idea of giving a better overview of how the system works. The class diagrams show the connection between the different components and how they interact, but they do not give a good idea of the flow from the user's perspective.

Browse to Webshop

The following sequence diagram (Figure 3.10) shows how the user navigates to the webshop while browsing through the information for a given product.

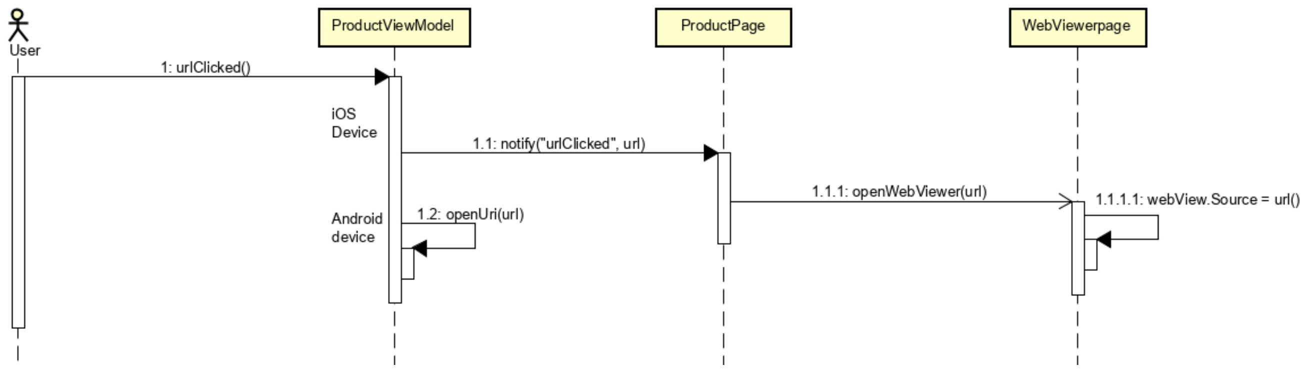


Figure 3.10 – Browse to Webshop Sequence

IMPORTANT NOTE: When the ProductViewModel calls "openUrl(url)" the user device will open the link (the url) in the default internet browser, thus it is not handled in the HB Products Application.

Browse Products – Error handling

The error handling is a vital part of each software product. The usability and user experience hardly rely on error handling. The following diagram Figure 3.11 represents how the application handles the cases which result in a failed request for a list of products. The system catches the error and displays it to the User in a pop-up message.

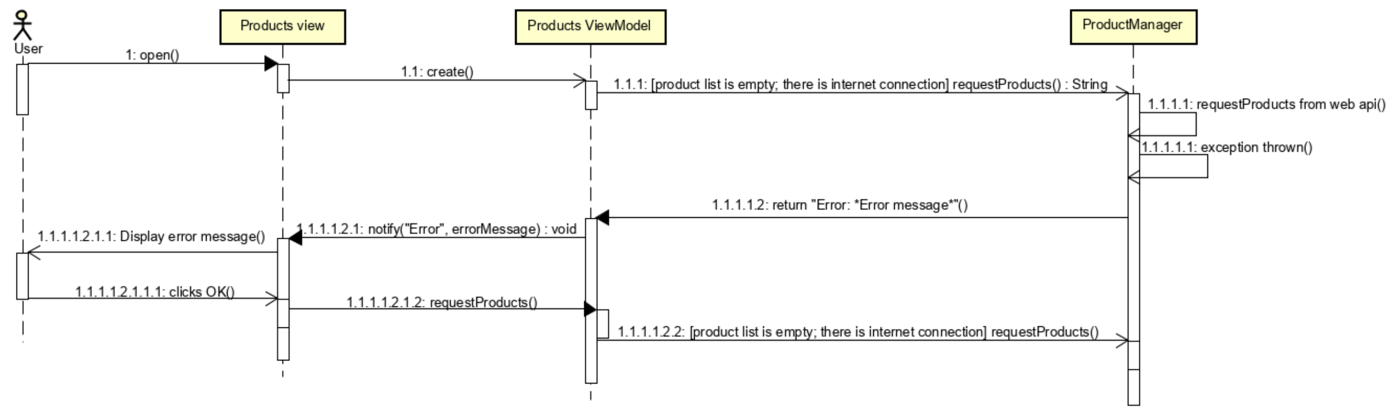

User
Products view
Products ViewModel
ProductManager
1: open()
1.1: create()
1.1.1: [product list is empty; there is internet connection] requestProducts() : String
1.1.1.1: requestProducts from web api()
1.1.1.1.1: exception thrown()
1.1.1.1.2: return "Error: *Error message*"()
1.1.1.1.2.1: notify("Error", errorMessage) : void
1.1.1.1.2.1.1: Display error message()
1.1.1.1.2.1.1.1: clicks OK()
1.1.1.1.2.1.2: requestProducts()
1.1.1.1.2.2: [product list is empty; there is internet connection] requestProducts()

Figure 3.11 – Exception Sequence Diagram – Failed Request

Because of the size of the other sequence diagrams for this system, they will not be displayed in this report. To see all sequence diagrams and the extra information about them please check Appendix G.

3.6.4 Deployment diagram

The shown deployment diagram in Figure 3.12 visualizes the hardware and software parts of the system separated into nodes. The HB Products application uses a 3-tier architecture. It contains a Client, Business Layer, and Data Layers. This diagram is a representation of the run-time architecture of the system and contains how the software elements are encapsulated into nodes.

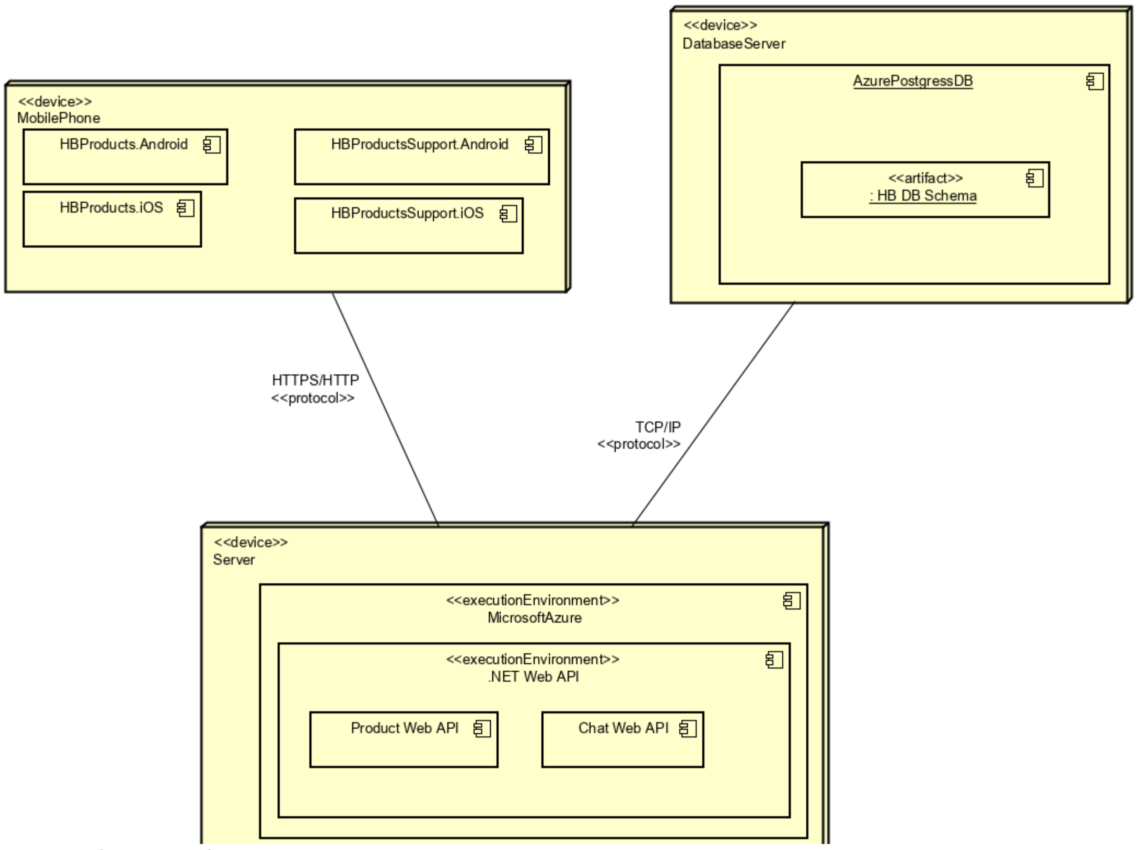


Figure 3.12 – System Deployment Diagram

The connections between the nodes illustrate the communication path and protocol used by the different hardware elements.

In this case, the system is a mobile application that connects to a .NET Web API using an HTTP Request. Then the Server which is connected to a database translates this request into a query and sends it through a TCP/IP protocol that returns the desired data back to the mobile phone.

3.6.5 ER Diagram

The Conceptual ER diagram is a diagram used for giving a better understanding of the connection between the objects in the database. Entities and relationships are modeled around the business' needs. Only Conceptual ERD supports generalization, which is reducing the size of the diagram and making it more readable. The figure below (Figure 3.13) represents the Conceptual ER diagram used for developing the database for storing the product data and the additional information for each product. Using this approach at any time the administrator can add additional data for each sensor without having to change any of the table structures.

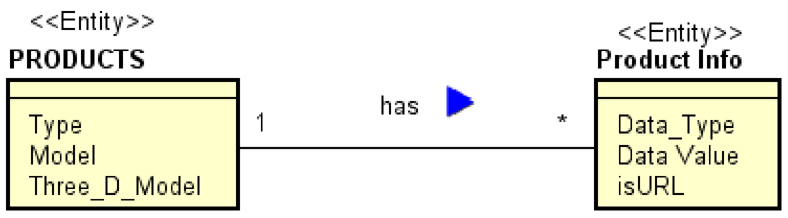


Figure 3.13 – Conceptual ER Diagram

Figure 3.14 is the ER Diagram of the database used for storing products and additional product information. ER Diagrams are used for visualizing the structure of each table in the database and the relationships between each table. The design of this diagram was done using Normalization for a database design technique so that redundancies and dependency of data are reduced. (Guru99, 2019) The database was designed to cover all rules required for the Third Normal Form (3NF). By doing this, as mentioned above, the database dependency and redundancies of data were reduced, which decreases the amount of data and the chance for making a mistake when inserting new data.

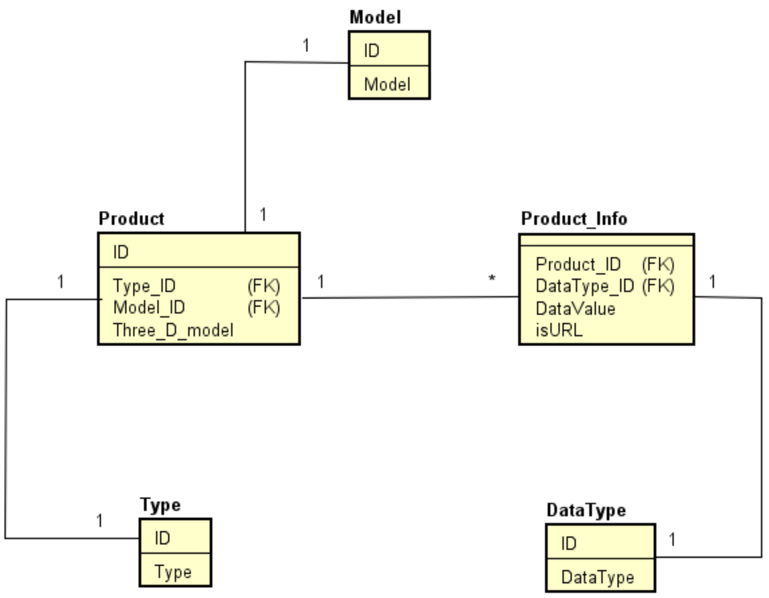


Figure 3.14 – ER Diagram

Note: The Product and Chat ER diagrams and Conceptual ER diagrams can be seen in Appendix I – ER Diagrams

4 Implementation

This part presents code pieces from the project and describes them in depth. It also contains an explanation of the complete path thought the system separated into its architecture layers.

4.1 Presentation Tier

4.1.1 3D Model View

The product owner wanted the 3D scene to be visible inside the application so that the customer does not leave the application when he/she selects to see a 3D representation of a selected product. This was a complicated task but after doing the research the team found out that the Urho Scene can be wrapped in a UrhoSurface and presented to the customer inside a content page.

```
protected override async void OnAppearing()
{
    (Xamarin.Forms.Application.Current.MainPage as MainPage).IsGestureEnabled
= false;

    modelViewer = await urhoSurface.Show<ThreeDModelViewer>(new
ApplicationOptions("Materials")
    { Orientation = ApplicationOptions.OrientationType.Portrait });

      modelViewer.setThreeDModelName(threeDModel);
      Urho.Application.InvokeOnMain(()=>modelViewer.startDisplaying());
}
```

The slide menu gesture had to be disabled while presenting the 3D model because it was conflicting with its touch input and the customer was not able to interact with the sensor. Disabling the sliding menu does not provide any negative side effects.

4.1.2 QR code scanner

One of the main ideas for the product owner was to provide their customers with easier access to information about a certain product. This should be achieved by scanning QR codes located on each of their products and present the users with additional information about the product.
To implement this feature into the application the development team used an open-source library called “ZXing.Net.Mobile” (Zebra Crossing). An open-source library is always useful but does not solve the problems for implementation. Since there are many security features on the newest phones, the user must allow the application to access the device’s integrated camera module and use it as a scanner. Then comes the next problem – each device platform (Android, iOS) uses a different methodology for granting permissions to the application. Fortunately, there is an open-source library for making cross-platform permission requests called “Plugin. Permissions”.
After describing the tools required to implement the requirement for the QR scanner, let's move on to the coding part.
Since the user experience is important for every project, the application tries to take some of the work away from the user. The application makes a scanner request as soon as the page is opened:

```
public ScanPage()
        {
            InitializeComponent();
            …
            RequestScanner();
        }
```

The RequestScanner() method checks if the application has permissions granted and starts the QR scanner if permissions are granted. If there are no permissions granted the system will prompt the user for permissions.

```
var status = await
CrossPermissions.Current.CheckPermissionStatusAsync<CameraPermission>();
if (status != PermissionStatus.Granted)
{
     if (await
CrossPermissions.Current.ShouldShowRequestPermissionRationaleAsync(Permission.
Camera))
     {
          await DisplayAlert("Need Camera", "In order to scan we will need
camera permissions", "OK");
     }

     status = await
CrossPermissions.Current.RequestPermissionAsync<CameraPermission>();
}
```

The code shown above is executed asynchronously so the system awaits for the user to answer and immediately after that it takes action according to the user's permission response.

```
if (status == PermissionStatus.Granted)
{
    //Permission granted
    StartScanner();
}
else if (status != PermissionStatus.Unknown)
{
    //Permission denied
    switch(Device.RuntimePlatform)
    {
        case Device.iOS:
             await DisplayAlert("Camera permissions denied!", "In order to
scan we will need camera permissions. Turn them on from the settings page
of our app.", "OK");
        break;
     }
}
```

As shown in the code above – if the system is granted permissions it starts the QR scanner. If not granted permissions the system checks what is the device platform, because the different systems handle the permissions differently – Android allows applications to ask for permissions multiple times, while on the other hand iOS only allows one permission request. If the permissions request is rejected the user has to manually navigate to the permission settings of the app in the settings menu and grant permissions from there. This is why on iOS the system prompts instructional message of how to grant permissions if they are denied.

Talking about the scanner few things should be mentioned – The ZXing library pushes another page with the camera on it on top of the navigation stack. After there is a barcode scanned the page is destroyed and it provides the system with a "ZXing.Result" object which has a "Text" parameter inside. This parameter gives a human-readable string representation of the scanned code.

```
ScannerPage.OnScanResult += (result) => { …handling the result here… }
```

After the results are gathered, the system checks if the scanned code represents a numerical ID of a product.

```
if(int.TryParse(result.Text, out scannedId))
```

If the result is an integer, then the system makes a request to the API to get information about a product with the scanned ID.

```
Device.BeginInvokeOnMainThread(() =>
                    {
                        Navigation.PopAsync();
                        getProductWithId(scannedId);
                    });
```

NOTE: The "Navigation.PopAsync();" method should be always called from the "UI Thread". The ScannerPage does not run on the main thread, therefore "Device.BeginInvokeOnMainThread()" is required in that case.

If a product with this ID exists in the database, the system is provided with information for the given product and the user is provided with a new ProductPage.

```
Product scannedProduct =
JsonConvert.DeserializeObject<Product>(productString);
startProductPage(scannedProduct);
```

In case that any error has occurred during those steps (product does not exist in the database, connectivity problem, etc.), the system displays a message with an explanation of what failed during the steps.

4.1.3 Model-View-View Model (MVVM) and data binding

As mentioned in section 3.6.1 it is very important to keep the code clean, scalable and understandable by following design patterns. In this section, only a selected number of cases for the usage of the MVVM design pattern will be shown.
A good example of how the MVVM design pattern and data binding are used in this project will be the page in which you browse the products and the page for product information.

To display a page with products, the system needs a list of products which are acquired by making a request to the API.

-How does the ProductPage get information about the currently available products and display them in the UI?
The answer is simple – the ProductPage creates a ProductsViewModel object, passing itself as a notifiable view, and binds the UI properties to that ViewModel. Indeed, that is the only job of a UI class in order to make a list of an unknown number of objects.

```
        public ProductsPage()
        {
            InitializeComponent();
            viewmodel = new ProductsViewModel(this);
            //Binding ViewModel to View...
            BindingContext = viewmodel;
        }
```

Every time that this page is opened it checks if there are any products loaded in the view model, and if there are none – it makes the view-model to make a request.

```
if (viewmodel.ProductList.ProductsCount() <= 0) //If the page is not filled
with products - request products.
                Task.Factory.StartNew(() => viewmodel.requestProducts());
```

The request is done on a separate thread from the UI to prevent stuttering UI elements and smooth UI flow.

When the ProductList property is updated, the view is notified by the OnPropertyChanged method in the View-Model.

```
        public ProductList ProductList
        {
            set { SetProperty(ref productList, value);
                 OnPropertyChanged("ProductList");
            }
            get { return productList; }
        }
```

After the ProductList property has been updated, the list in the View will automatically update its fields and generate product objects in the list.

```
<StackLayout BindingContext ="{Binding ProductList}">
   <ListView ItemsSource="{Binding Products}"
                   ItemTapped="OnItemSelected"
                   …Some properties…
                   IsRefreshing="{Binding BindingContext.IsLoading,
                                   Source={x:Reference parentView}}"
                   RefreshCommand="{Binding BindingContext.RefreshProducts,
                                   Source={x:Reference parentView}}">
       <ListView.ItemTemplate>
          <DataTemplate>
              <ImageCell ImageSource="{Binding ThumbNailSource}"
                         Text="{Binding Model}"
                         TextColor="{x:DynamicResource HB}"
                         Detail="{Binding Type}"/>

    …Closing tags…
```

In the code snippet above the binding of the Products list is shown. The StackLayout is binding onto the ProductList from the View Model which contains all of the products that were retrieved from the API. The StackLayout's child is a ListView that has the ItemsSource property binding onto the Products property in the ProductList. The Products property is an observable collection of Product objects. The Products property is not in the View Model class but the ListView can bind to it because it inherits the properties of its parent's binding – Products, which is a bindable property in the ProductList class. In the same way, the ImageCell binds to the properties of its parent's binding – each item from "Products" is a Product object which contains bindable properties such as Type, Model, ThumbnailSource and others. In this context, they are the only ones that are required. To get a better representation of how the UI looks with a few items drawn onto the ListView see Figure 4.1.

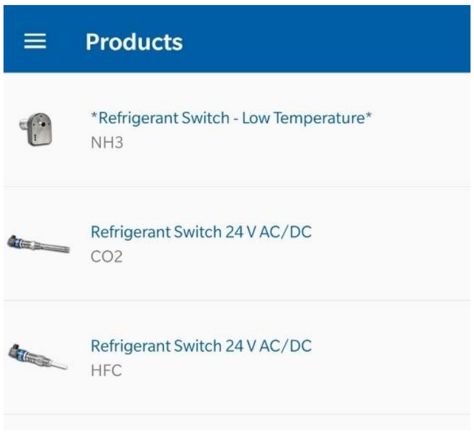

Products
Refrigerant Switch - Low Temperature
NH3
Refrigerant Switch 24 V AC/DC
CO2
Refrigerant Switch 24 V AC/DC
HFC

Figure 4.1 – UI List View

The "ItemTapped" property from the product list specifies what happens when an item from the list is clicked. In this case, it calls a method "OnItemSelected" in the view. Then the view notifies the View-Model that a product was clicked and sends the selected product reference as an argument – see code snippet below.

```
private async void OnItemSelected(object sender, ItemTappedEventArgs e)
{
    productList.SelectedItem = null;
    if (!viewmodel.NoInternetConnection)
    {
         //There is internet connection - make a new thread for the request
         Task.Factory.StartNew(() => productClicked(e));
    }else
    {
         //There is no internet connection
         await DisplayAlert("Alert", "Turn on internet connectivity services.", "OK");
    }
}
```

Then after that, the ViewModel gets information about the selected product from the API and starts a new ProductPage if everything was successful.

IMPORTANT NOTE: The full source code could be found in Appendix H

ProductPage

The product Page has a UI consisting of an image, text data, and buttons.

```
<Image Source="{Binding ProductImage}" …some more properties…/>
<!--Makes a list of all of the data which is NOT containing URLs(e.g. Product
Description)-->
           <ListView ItemsSource="{Binding NoURLData}"
                        HasUnevenRows="True"
                          x:Name="listViewNoURLData">
              <ListView.ItemTemplate>
                <DataTemplate>
                   <ViewCell IsEnabled="False">
                      <StackLayout>
                        <Label Text="{Binding Type}" …some more properties…/>
                        <Label Text="{Binding Value}" …some more properties…/>
                  …closing tags…
```

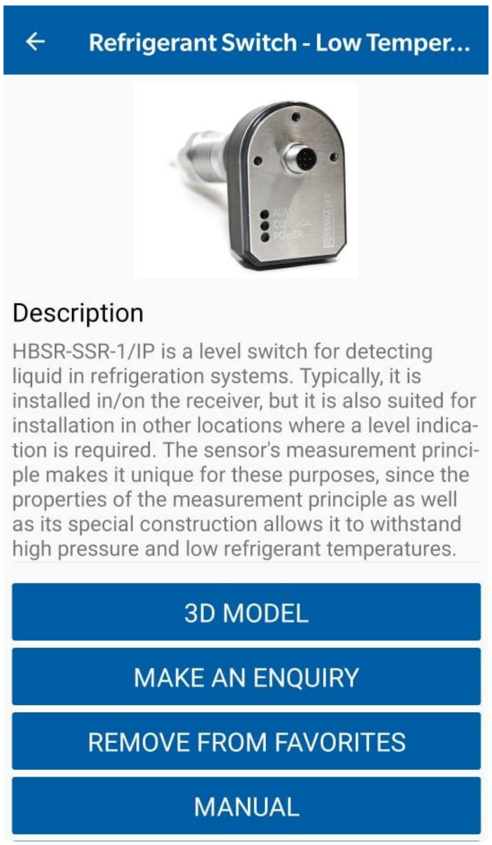
First is the product image and then a list of "NoURLData". The "NoURLData" is a list consisting of ProductData objects that are not URLs's e.g. Product description.

Then come some buttons that are typical for each product – Mark as a favorite, 3D model, make an enquiry.

After the typical buttons that every product has, the buttons that are dynamically generated come. They are generated based on the ProductData objects that are urls.

```
<ListView ItemsSource="{Binding URLData}" …some more properties…>
      <ListView.ItemTemplate>
           <DataTemplate>
                <ViewCell>
                   <StackLayout Margin="0,0,0,5" >
                       <Button
                           Text="{Binding Type}"
                           FontSize="Large"
                           TextColor="White"
                           Command="{Binding BindingContext.URLClicked,
                                  Source={x:Reference ProductPageView}}"
                           CommandParameter="{Binding Value}"/>
        …closing tags…
```

Each button's Text property is binding to the type of ProductData explaining what the button refers to (e.g. Quick Guide, Manual, etc.). The command is what is called after the button is clicked. This is the part where it gets tricky since it is required that the Text field bind to a property of the ProductData class and the Command property to bind to a property in the view model. To achieve that, the source of the command bindable property is referencing the ProductPageView which is the root view element of the page – it's binding to the view model. The "CommandParameter" property sends a parameter to the command called in the View Model which is the "Value" property of each ProductData object which represents the link that the user should be redirected to. By implementing this View the following design look is achieved - see Figure 4.2



Description

HBSR-SSR-1/IP is a level switch for detecting liquid in refrigeration systems. Typically, it is installed in/on the receiver, but it is also suited for installation in other locations where a level indication is required. The sensor's measurement principle makes it unique for these purposes, since the properties of the measurement principle as well as its special construction allows it to withstand high pressure and low refrigerant temperatures.

Figure 4.2 – Product Data View

4.2 Logic Tier

4.2.1 Model-View-View Model and Data binding

In this section, the "view model" part of the design pattern will be reviewed. The following code snippet shows how a request is made to the API from the ProductsViewModel class, using the ProductManager.

```
//Requests the products from the product manager.
        public async void requestProducts()
        {
            //Check if there is internet connection
            if (Connectivity.NetworkAccess == NetworkAccess.Internet)
            {
                NoInternetConnection = false;
                IsLoading = true;
                string productListString = await manager.requestProductsAsync();
                IsLoading = false;

            … Error handling…

            …Sorting…
            } else {
                NoInternetConnection = true;
            }
        }
```

When the product list is received from the Products Manager it gets sorted before being assigned to the bindable property. This is done so that the favorite products come on top of the list and have asterisk signs in front and back.

```
//Set the sorted ProductList variable to a sorted product list so that the
favorites come on top.
                ProductList =
SortProductList(JsonConvert.DeserializeObject<ProductList>(productListString));
```

The sorting is done by changing the places of the top items of the list with the ones that are marked as favorite by the user.

```
//Sorts the product list so that it puts the favorites on top.
     private ProductList SortProductList(ProductList products)
     {
         List<int> favProducts = Settings.GetFavoriteProducts();
         if (favProducts.Count == 0) //Check if there are any favorite products...
             return products;

         int favIndex = 0;

         for (int i = 0; i < products.ProductsCount(); i++)
         {
             if (favProducts.Contains(products.Products[i].Id))
             {
                 products.Products[i].Model = "*" + products.Products[i].Model + "*";
                 Product temp = products.Products[favIndex];
                 products.Products[favIndex] = products.Products[i];
                 products.Products[i] = temp;
                 favIndex++;
             }
         }
         return products;
     }
```

By doing all of the operations in the view model, the UI thread is not overloaded, and the UI is smooth and clean. This approach also decouples the system's layers. Let's say for example in the future the logic on how to retrieve the product list is changed. The only changes will be in the view model and the product manager. The UI will be untouched and will display the products in the same way to the user since it does not take care of requests to the API, error handling, etc.
IMPORTANT NOTE: The full source code could be found in Appendix H

4.2.2 Customer support chat

In this section, the customer support chat will be reviewed, explaining how the system updates the list of messages and sends messages. The user interface was integrated from an existing project from GitHub. (Phillpotts, 2018)

Firstly, it should be mentioned that the chat in the customer and the employee app are virtually the same with the small difference that the customer listens for new messages from the employee, and vice versa.

Getting into a chat session

Once again it should be mentioned that for the development team and the product owner, one of the most important aspects of every computer software is customer satisfaction. So, the chat was designed in a way that it would first check if you have ongoing chat sessions before opening a new one. This prevents sessions to be lost when the app was closed, or the customer's device fails for any reason. The first thing that happens after entering the required information from the user (name and email) is to get a session ID from the API. The API checks the database for existing sessions with this user and returns the session id of the existing session. If a session with this user is non-existent, a new session is created, and the ID is returned to the application. After the session id is retrieved from the API, the application pushes a new ChatPage on the navigation stack.

```
    int sessionRes = await manager.GetSesionId(email.Text, fullName.Text);
            …error handling in case the request failed…
    await Navigation.PushAsync(new ChatPage(sessionRes));
```

Having the ChatPage open, the system needs to check if there are any existing messages in that session. This is done in the view model class by the method "SetSession". This method gets called inside of the constructor, so it happens immediately after the view model object is created.

```
private async void SetSession(int sessionID)
{
       … some error handling…

    string chatString = await manager.GetSessionInfo(sessionID);
    this.chat = JsonConvert.DeserializeObject<Session>(chatString);
    lastMessageID = chat.GetLatestEmployeeMessageID(); //Update the latest message id
so that the message does not appear multiple times.

    if (chat.MessageList != null) {
        foreach (Message m in chat.MessageList) //Add all previous messages to the
chat page.
    Messages.Add(new TextChatViewModel() { Text = m.Text, Direction = m.IsEmployee ?
TextChatViewModel.ChatDirection.Incoming : TextChatViewModel.ChatDirection.Outgoing
});
    view.notify("new messages"); //Notify the view so that it scrolls to the latest
message in the chat session.
    }
}
```

If there are any messages existing in the session retrieved from the API, the view model adds them to the Messages list and notifies the view for the changes, so that the view can then scroll to the last item in the chat.

Sending a message

Sending a message is done when the user has entered some text in the entry field and has pressed the “Send” button next to it. When those steps are done the view model takes the text from the entry field, and makes a message object containing the text. The message object is then passed to the chat manager to send it to the API. If the response is successful, the message is added to the message list and the view is notified for new messages (the UI scrolls back to the bottom of the message list).

Error handling in the case of message sending requires some special treatment in order to provide a better user experience. If more than one message is being sent in a short time interval, the UI is going to be filled with error pop-up messages if such a message is displayed each time. This is why the system allows up to 4 failed attempts to send a message before actually displaying the error (see code snippet below).

```
case -200: //Http request returned code different from "200 OK"
          if (failedMessages > 4)
          {
            view.notify("message error", "Internal server error"); //Too many
failed attempts
            return;
          }
          failedMessages++;
          SubmitMessage(messageText); //make another attempt to send the
message.
          return;
```

Receiving messages

To receive messages the application sends a request to the server every 3 seconds. This method is recognized as "Short polling" and compared to "Long polling" it sends update requests on fixed time intervals instead of waiting for the server to notify for any changes in the data set (see code snippet below).

```
Task.Factory.StartNew(() =>
            {
                //Start a timer which requests new messages every 3 seconds.
                if (manager == null || chat == null) return;
                var startTimeSpan = TimeSpan.Zero;
                var periodTimeSpan = TimeSpan.FromSeconds(3);
                timer = new Timer((e) =>
                {
                    GetLatestMessages();
                }, null, startTimeSpan, periodTimeSpan);
```

As you can see in the code snippet the updating is done by a timer which is running on its own thread to keep the rest of the system responsive and well-performing. The "GetLatestMessages()" requests messages from the API which have higher message-id than the last one that the system currently has. In this way, the data transfer is brought to a minimum, while having the message list always up to date.

IMPORTANT NOTE: The full source code could be found in Appendix H

Send a chat copy

In order to send a chat copy, the system uses the SendGrid API that is also used for sending enquiries. The chat copy is done by requesting the session from the server and then running it through an algorithm that creates HTML code to represent the chat. The system requests the session from the server instead of using the local session object to prevent any possible errors.

The generated HTML code is afterward sent to the email that the user-specified when creating a session.

```
string chatString = await getChatString();
var msg = new SendGridMessage()
{
    From = new EmailAddress("chat-manager@hbapp.com",  "ChatMessenger"),
    Subject = "HB Products chat copy",
    PlainTextContent = "Chat Copy!",
    HtmlContent = chatString
 };

 msg.AddTo(new EmailAddress(chat.Customer.Email, chat.Customer.Name));
```

4.2.3 3D Rendering Engine

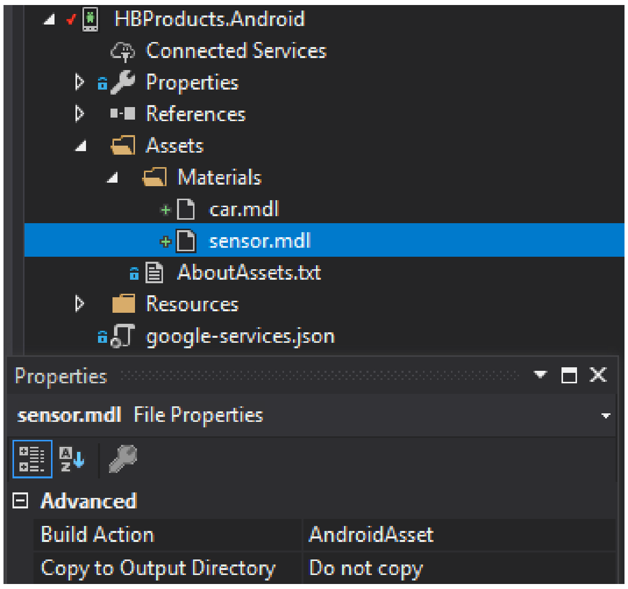
For fulfilling “The customer should be able to interact with the 3D model using touch inputs” a 3D engine needed to be implemented. After a lot of researching the choice of 3D engine for this project was UrhoSharp.

The engine implementation consists of a few classes and the UrhoSharp.Forms (Xamarin, 12/4/2018) NuGet Package. The ThreeDModelViewer class is responsible for the initialization of the engine, setting up the scene, setting up the sensor and the touch interaction with the sensor.

```
// 3D scene setup
            scene = new Scene(Context);
            scene.CreateComponent<Octree>();

            // Create a node for the Sensor
            sensorNode = scene.CreateChild();
            sensorNode.Position = new Vector3(0, 0, 5);
            sensorNode.Rotation = new Quaternion(5, 0, 10);
            sensorNode.SetScale(0.01f);

            // Create a static model component:
       StaticModel sensor = sensorNode.CreateComponent<StaticModel>();
       sensor.Model = ResourceCache.GetModel(threeDModelName); //Load the
model

            // Light
            Node lightNode = scene.CreateChild();
            var light = lightNode.CreateComponent<Light>();
            light.LightType = LightType.Directional;
            light.Range = 20;
            light.Brightness = 1f;
            lightNode.SetDirection(new Vector3(0.4f, -0.5f, 0.3f));

            // Camera
            cameraNode = scene.CreateChild();
            var camera = cameraNode.CreateComponent<Camera>();

            // Viewport
            var viewport = new Viewport(Context, scene, camera, null);
            Renderer.SetViewport(0, viewport);
        }
```

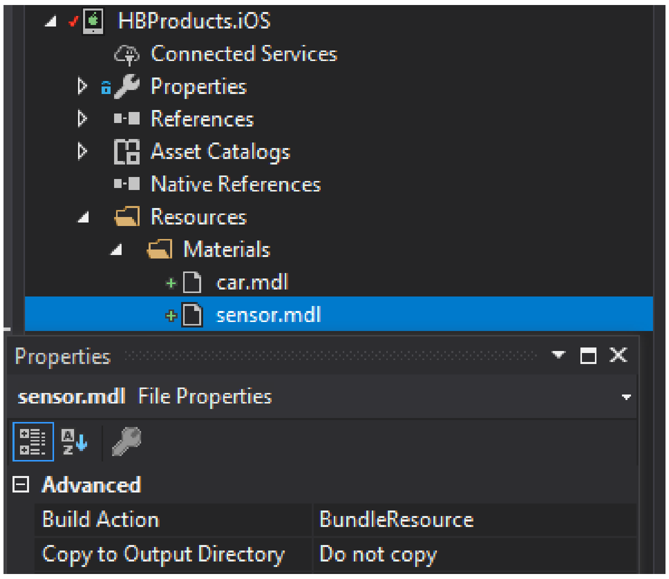
Setting up the Scene

The extension that UrhoSharp (Egor Bo, 2017) uses is “. mdl”. Unfortunately, when the company asked us in what format they should provide the 3D models “. mdl was not one of the options so the team decided to go with the “.obj” file which is a standard format that can be converted in “. mdl”. For the developer to be able to display the 3D sensor in UrhoSharp it first needs to be a single object and in “. mdl”. The provided 3D model was not a single object, so all its parts needed to be joined into one and then exported. For this part Blender was used, because UrhoSharp had a ”.mdl” asset exporter plugin for Blender (reattiva/Urho3D-Blender, 2019). After the 3D model was joined and exported in “. mdl” it needs to be put in both of the operating systems asset folders(Figure 4.3 and Figure 4.4) and their build actions must be set properly.


```
◢ HBProducts.Android
    Connected Services
  ▷ Properties
  ▷ References
  ◢ Assets
    ◢ Materials
        car.mdl
        sensor.mdl
      AboutAssets.txt
  ▷ Resources
    google-services.json
```
Properties
sensor.mdl File Properties

Advanced	
Build Action	AndroidAsset
Copy to Output Directory	Do not copy

Figure 4.3 – Android System Asset Folder


```
◢ HBProducts.iOS
    Connected Services
  ▷ Properties
  ▷ References
  ▷ Asset Catalogs
    Native References
  ◢ Resources
    ◢ Materials
        car.mdl
        sensor.mdl
```
Properties
sensor.mdl File Properties

Advanced	
Build Action	BundleResource
Copy to Output Directory	Do not copy

Figure 4.4 – iOS System Resources Folder

The MoveCameraByTouches method is called in the OnUpdate which is an UrhoSharp method which is called every frame by using this the engine always checks for touch inputs.

```
    protected void MoveCameraByTouches(float timeStep)
        {
            const float touchSensitivity = 200f;

            var input = Input;

            if (input.NumTouches > 0)
            {
                touch1 = input.GetTouch(0);

                if (input.NumTouches == 2) //Two touch inputs (user
zooming/unzooming the object)
                {
                    touch2 = input.GetTouch(1);
                    // Check for zoom pattern (touches moving in opposite
directions and on empty space)
                    if ((((touch1.Delta.Y > 0 && touch2.Delta.Y < 0) ||
(touch1.Delta.Y < 0 && touch2.Delta.Y > 0))))
                    {
   // Check for zoom direction (in/out)
                        if (Math.Abs(touch1.Position.Y - touch2.Position.Y) <
Math.Abs(touch1.LastPosition.Y - touch2.LastPosition.Y))
                        {
                            if (!(scale <= MinScale))
                            {
                                //Unzoom
                                scale -= ZoomSpeed;
                                sensorNode.SetScale(scale);
                            }
                        }
                        else
                        {
                            if (!(scale >= MaxScale))
                            {
                                //Zoom
                                scale += ZoomSpeed;
                                sensorNode.SetScale(scale);
                            }
                        }

                    }
                }



                else if (input.NumTouches == 1) //Only one touch input (user
rotating the object)
                {
                    if (touch1.Delta.X != 0 || touch1.Delta.Y != 0)
                    {
                        var camera = sensorNode.GetComponent<StaticModel>();
                        if (camera == null)
                            return;

                        x += touchSensitivity / Graphics.Height * touch1.Delta.X;
                        y += touchSensitivity / Graphics.Height * touch1.Delta.Y;


                        sensorNode.Rotation = new Quaternion(y, x, 0);
                    }
                }

            }

        }
```

Touch Camera Movement and Zoming

4.2.4 Push Notification Service

One of the project’s requirements was **“The product owner should be able to send push notifications to the users of the app.”** Therefore, a push notification service was implemented using NuGet Package “Plugin.AzurePushNotification” (CrossGeeks, 2019). This notification plugin was chosen because it was cross-platform compatible and worked with Azure Notification Hub.
The Plugin needed to be implemented into both platforms natively using their own proprietary Notification Service.

Set-Up

Android

For the Android Notification to work. The application had to be connected with Firebase. Google Services and FCM (Firebase Cloud Messaging) had to be added to the app and an API token was taken from Firebase and put in Azure Notification Hub.

iOS

For Apple Notification service to work. The team needed an Apple Developer Account because of some limitations that Apple applies when a project is developed without one. The Apple Developer Account was provided to the team by the team’s supervisor. After acquiring the Apple Developer Account, the application had to be registered for APNS (Apple Push Notification service). After registering the application for this service an application-specific certificate was issued by Apple and it was put in the Azure Notification Hub.

Implementation

Android

As mention before the Plugin that was used to implement the Notifications required its implementation natively. Few classes needed to be created so that the notification service functioned properly. At first, it was only initializing the FCM (Firebase Cloud Messaging) and the application received notifications only while it was in the foreground.

```
//Set the default notification channel for your app when running Android Oreo
if (Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.O)
{
   //Change for your default notification channel id here
   AzurePushNotificationManager.DefaultNotificationChannelId =
"DefaultChannel";

   //Change for your default notification channel name here
   AzurePushNotificationManager.DefaultNotificationChannelName = "General";
}

AzurePushNotificationManager.Initialize(this,

Constants.ListenConnectionString,Constants.NotificationHubName, true);
```

After that, a splash activity was added and the notification service was moved out from the main activity to its own application class so that it can run as a background service and open the app when a notification is clicked.

```
// Main Activity is put here to avoid issues while opening the app by pressing
the notification
 var mainIntent = new Intent(Application.Context, typeof(MainActivity));

 if (Intent.Extras != null)
 {
    mainIntent.PutExtras(Intent.Extras);
 }
 mainIntent.SetFlags(ActivityFlags.SingleTop);

 StartActivity(mainIntent);
```

iOS

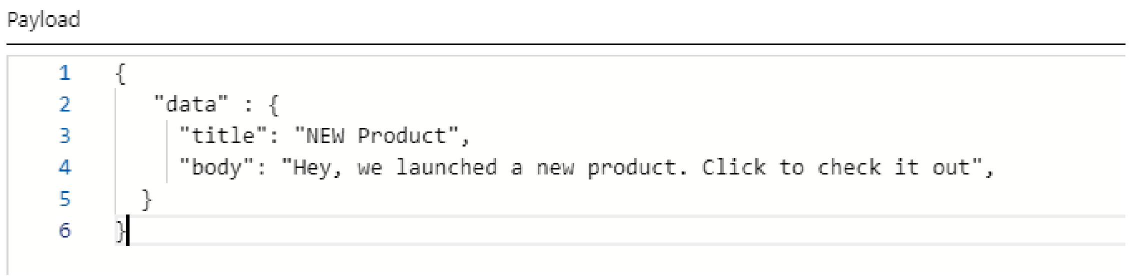
The Implementation on the iOS part was simpler because registering the application for APNS automatically set it up in a way that it was receiving notifications in the background. Here the Notification Plugin had to only be initialized and some native methods had to be overridden.

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
        {
            LoadApplication(new App());

AzurePushNotificationManager.Initialize(Constants.ListenConnectionString,
Constants.NotificationHubName, options, true) ;
            AzurePushNotificationManager.CurrentNotificationPresentationOption =
UNNotificationPresentationOptions.Alert |
UNNotificationPresentationOptions.Badge;
            CrossAzurePushNotification.Current.RegisterAsync(new string[] {
"ios", "general" });

            return base.FinishedLaunching(app, options);
        }
        public override void RegisteredForRemoteNotifications(UIApplication
application, NSData deviceToken)
        {

AzurePushNotificationManager.DidRegisterRemoteNotifications(deviceToken);
        }

        public override void FailedToRegisterForRemoteNotifications(UIApplication
application, NSError error)
        {

AzurePushNotificationManager.RemoteNotificationRegistrationFailed(error);

        }
        // To receive notifications in foregroung on iOS 9 and below.
        // To receive notifications in background in any iOS version
        public override void DidReceiveRemoteNotification(UIApplication
application, NSDictionary userInfo, Action<UIBackgroundFetchResult>
completionHandler)
        {

            AzurePushNotificationManager.DidReceiveMessage(userInfo);
        }
    }
```

Notification Format

The notification payload format (See Figure 4.5 and 4.6) for the different platforms is also different.

Android

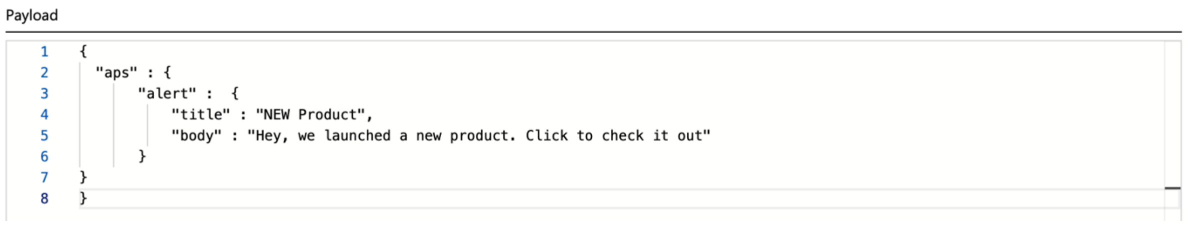


Payload

```
{
   "data" : {
     "title": "NEW Product",
     "body": "Hey, we launched a new product. Click to check it out",
  }
}
```

Figure 4.5 – Android Notification Payload Format

iOS



Payload

```
{
  "aps" : {
      "alert" :  {
          "title" : "NEW Product",
          "body" : "Hey, we launched a new product. Click to check it out"
      }
  }
}
```

Figure 4.6 – iOS Notification Payload Format

To both of the platforms, a specific TAG was added so that all the registered devices receive the notification. This TAG has to be written in the “Send to Tag Expression” in the Azure Notification Hub otherwise notification will not be pushed to registered devices.

TAG iOS - [ios]

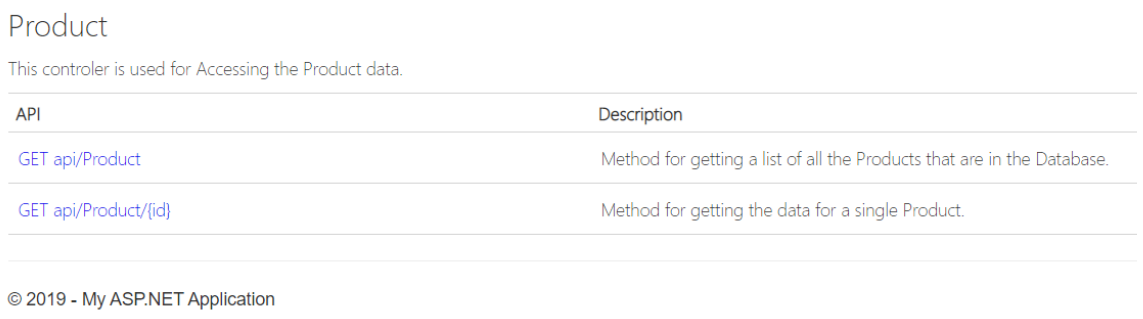
```
CrossAzurePushNotification.Current.RegisterAsync(new string[] { "ios", "general" });
```

TAG Android – [Android]

```
CrossAzurePushNotification.Current.RegisterAsync(new string[] { "android", "general"
});
```

4.2.5 WEB API

In this project, two WEB API servers were created. The first one is used for getting all the information about the company products and the second one is used for handling the data from the chat system. The choice for two APIs was made so that each server has a single responsibility and the failure of one does not influence the functionality of the other one.



Product

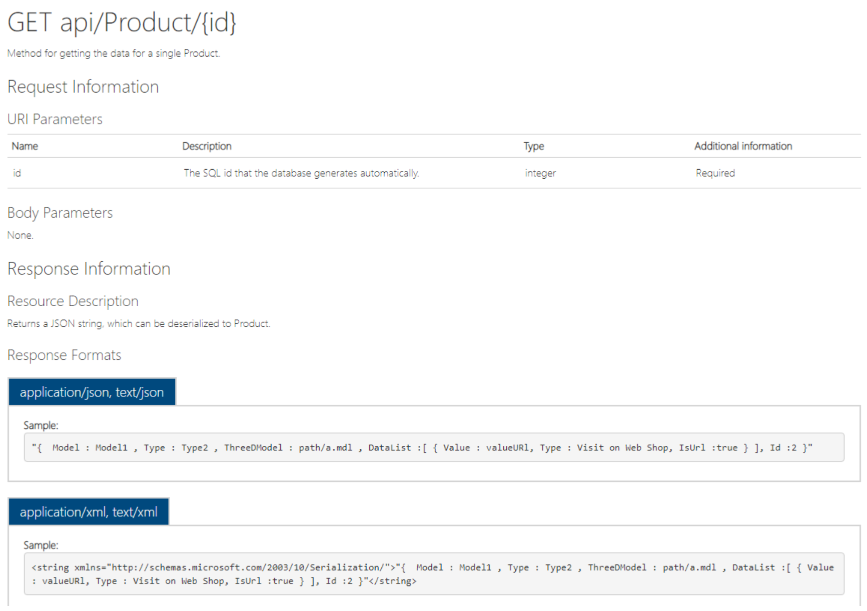
This controler is used for Accessing the Product data.

API	Description
GET api/Product	Method for getting a list of all the Products that are in the Database.
GET api/Product/{id}	Method for getting the data for a single Product.

© 2019 - My ASP.NET Application

Figure 4.7 – API Documentation

On the Figure 4.7 can be seen all the methods which are in the API, responsible for giving the product information, with a description for each one. When clicking on a certain method more detail description is shown, this can be seen on the Figure 4.8



GET api/Product/{id}

Method for getting the data for a single Product.

Request Information

URI Parameters

Name	Description	Type	Additional information
id	The SQL id that the database generates automatically.	integer	Required

Body Parameters

None.

Response Information

Resource Description

Returns a JSON string, which can be deserialized to Product.

Response Formats

application/json, text/json

Sample:

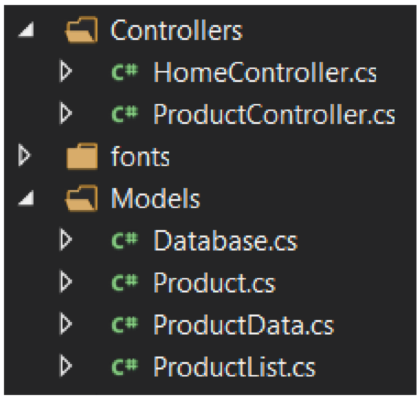
```
"{  Model : Model1 , Type : Type2 , ThreeDModel : path/a.mdl , DataList :[ { Value : valueURl, Type : Visit on Web Shop, IsUrl :true } ], Id :2 }"
```

application/xml, text/xml

Sample:

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">"{  Model : Model1 , Type : Type2 , ThreeDModel : path/a.mdl , DataList :[ { Value
: valueURl, Type : Visit on Web Shop, IsUrl :true } ], Id :2 }"</string>
```

Figure 4.8 – Detailed API Method Documentation



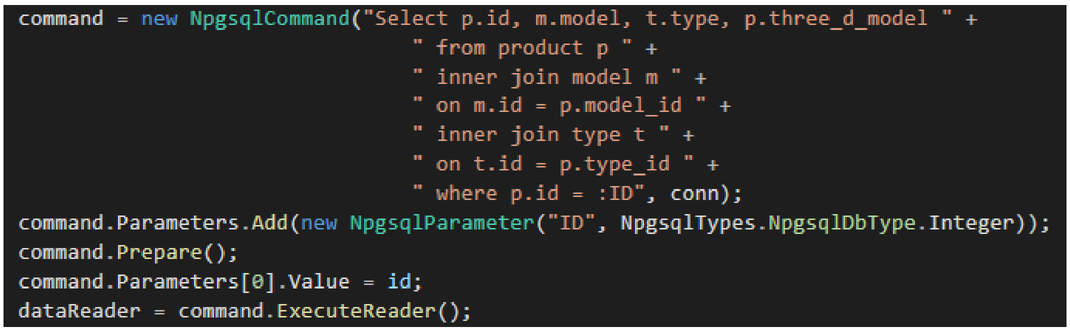
```
Controllers
    HomeController.cs
    ProductController.cs
fonts
Models
    Database.cs
    Product.cs
    ProductData.cs
    ProductList.cs
```

Figure 4.9 – Controller and Model of WEB API

For the implementation of the WEB API, the default creation MVC pattern was used for separating the classes to a model and controllers (See Figure 4.9). The model consists of a Product, ProductData and ProductList classes that are used for making the objects, which are used in the system. The Database class is used for making the connection to the Database and executing the statements.

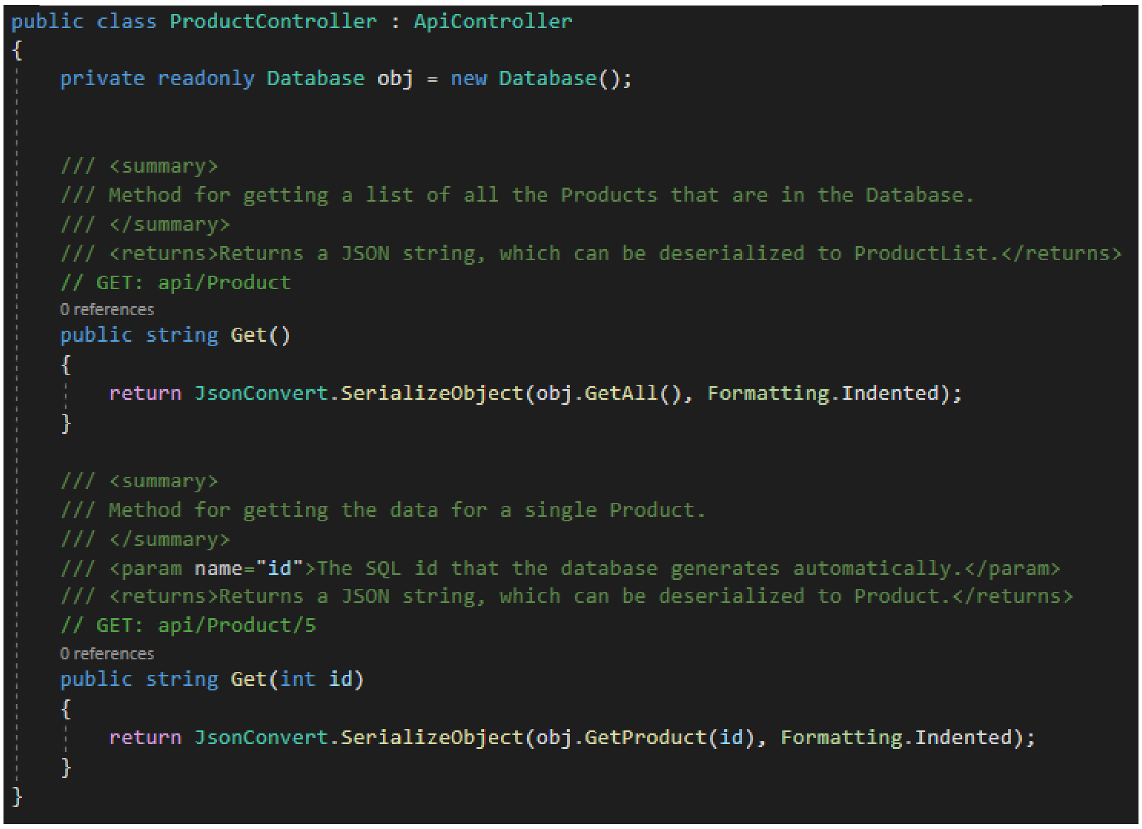
There are two ways of creating the statements, which are executed in the Database class. The first way is by going with a standard statement (Unprepared) and the second is with Prepared statements. Prepared statements were used because of the fact that they are faster to execute, especially when the number of table joins increases. These statements are faster because they are stored in the memory after the first execution and after that, the database system knows how to handle the statement instead of going the long way again. Another benefit of the prepared statements is that they are removing the possibility of making a SQL injection (SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution). (Wikipedia - SQL Injection, 2019) An example of a prepared statement can be seen in Figure 4.10.

IMPORTANT NOTE: The full source code could be found in Appendix H



```
command = new NpgsqlCommand("Select p.id, m.model, t.type, p.three_d_model " +
                            " from product p " +
                            " inner join model m " +
                            " on m.id = p.model_id " +
                            " inner join type t " +
                            " on t.id = p.type_id " +
                            " where p.id = :ID", conn);
command.Parameters.Add(new NpgsqlParameter("ID", NpgsqlTypes.NpgsqlDbType.Integer));
command.Prepare();
command.Parameters[0].Value = id;
dataReader = command.ExecuteReader();
```

Figure 4.10 – Prepared Statement



```
public class ProductController : ApiController
{
    private readonly Database obj = new Database();


    /// <summary>
    /// Method for getting a list of all the Products that are in the Database.
    /// </summary>
    /// <returns>Returns a JSON string, which can be deserialized to ProductList.</returns>
    // GET: api/Product
    0 references
    public string Get()
    {
        return JsonConvert.SerializeObject(obj.GetAll(), Formatting.Indented);
    }

    /// <summary>
    /// Method for getting the data for a single Product.
    /// </summary>
    /// <param name="id">The SQL id that the database generates automatically.</param>
    /// <returns>Returns a JSON string, which can be deserialized to Product.</returns>
    // GET: api/Product/5
    0 references
    public string Get(int id)
    {
        return JsonConvert.SerializeObject(obj.GetProduct(id), Formatting.Indented);
    }
}
```

Figure 4.11 – Web API Methods

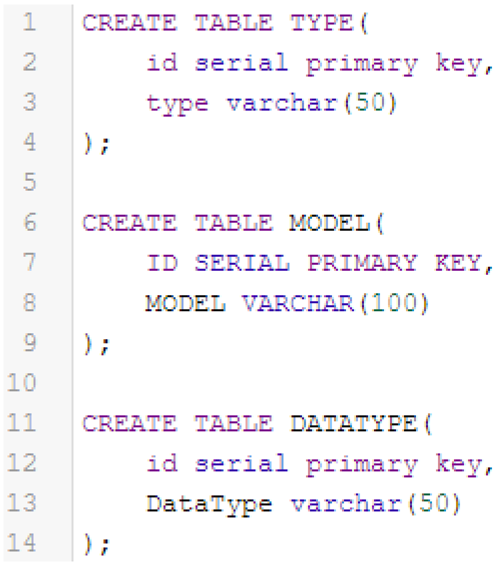
Figure 4.11 is the code written in the .NET WEB Application project for the two methods shown on Figure 4.7. Both methods are using an instance of the Database class to get the information from the database and after getting the information, the server is serializing the response from the database to a string using JSON and passing it as a return value.

IMPORTANT NOTE: The full source code could be found in Appendix H

4.3 Data Tier

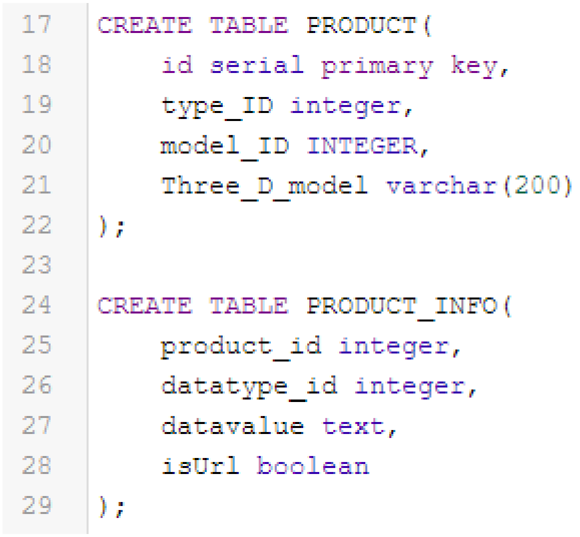
4.3.1 Product Database

For storing the product information, five tables were created. The script for creating the table can be seen on figure 4.12 and figure 4.13. The relations between the table were made by the script shown on figure 4.14.



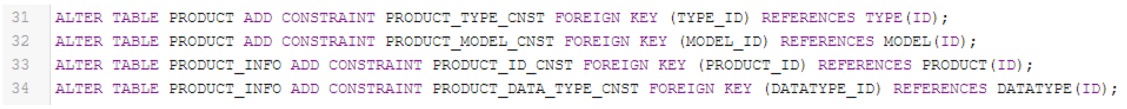
```
CREATE TABLE TYPE(
    id serial primary key,
    type varchar(50)
);

CREATE TABLE MODEL(
    ID SERIAL PRIMARY KEY,
    MODEL VARCHAR(100)
);

CREATE TABLE DATATYPE(
    id serial primary key,
    DataType varchar(50)
);
```

Figure 4.12 – Table Creation Script



```
CREATE TABLE PRODUCT(
    id serial primary key,
    type_ID integer,
    model_ID INTEGER,
    Three_D_model varchar(200)
);

CREATE TABLE PRODUCT_INFO(
    product_id integer,
    datatype_id integer,
    datavalue text,
    isUrl boolean
);
```

Figure 4.13 - Table Creation Script



```
ALTER TABLE PRODUCT ADD CONSTRAINT PRODUCT_TYPE_CNST FOREIGN KEY (TYPE_ID) REFERENCES TYPE(ID);
ALTER TABLE PRODUCT ADD CONSTRAINT PRODUCT_MODEL_CNST FOREIGN KEY (MODEL_ID) REFERENCES MODEL(ID);
ALTER TABLE PRODUCT_INFO ADD CONSTRAINT PRODUCT_ID_CNST FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT(ID);
ALTER TABLE PRODUCT_INFO ADD CONSTRAINT PRODUCT_DATA_TYPE_CNST FOREIGN KEY (DATATYPE_ID) REFERENCES DATATYPE(ID);
```

Figure 4.14 – Relation Creation

Table Type – used for storing all the types of products like “CO2” and “NH3”, which are stored in the type column. The id column of the table is a primary key and it is used for making the relationship with table Product.

Table Model – used for storing all the models of products like “Refrigerant Switch 24 V AC/DC” and “Refrigerant Switch - Low Temperature”, which are stored in the model column. The id column of the table is a primary key and it is used for making the relationship with table Product.

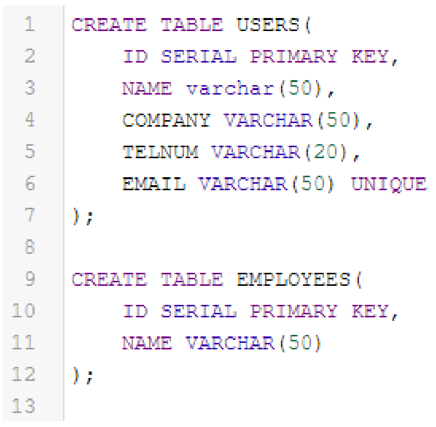
Table DataType - used for storing all the types of additional product data like "Image", "Thumbnail", "Visit on Web Shop", "Description", "Quick Guide" and "Manual", which are stored in the DataType column. The id column of the table is a primary key and it is used for making the relationship with table Product Info.

Table Product – used for storing the products. This table has a relationship with tables Type and Model. Each row of the table represents a product that has type id, which is connected to the Type table and model id, which is connect to the Model table, and because each sensor has only one 3D model the information about the product 3D model is saved in this table.

Table Product Info – used for storing additional product data. This table has a relationship with tables DataType and Product. Each row of this table is storing one additional information for a single product. The product id column is used for making the connection with the table Product so that the additional product data can be attached to a product. The DataType Id column is storing the connection of the table with table DataType, which is used for getting what is the type of the additional data. The dataValue column is storing the information of the additional product data which can be plain text or URL. To specify if the dataValue is text or URL the isURL column is used and this is a Boolean column, which holds true, if the dataValue is a URL or false if the dataValue is text.

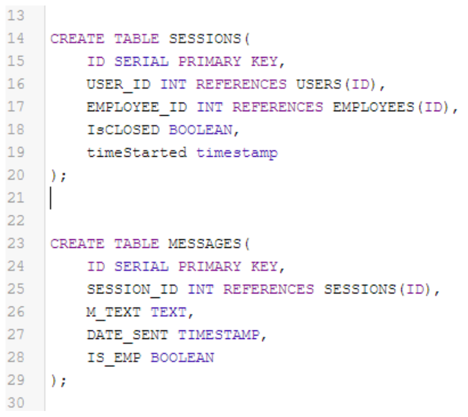
4.3.2 Chat Database

For storing the chat data, four tables were created. The script for creating the table can be seen on figure 4.15 and figure 4.16



```
CREATE TABLE USERS(
    ID SERIAL PRIMARY KEY,
    NAME varchar(50),
    COMPANY VARCHAR(50),
    TELNUM VARCHAR(20),
    EMAIL VARCHAR(50) UNIQUE
);

CREATE TABLE EMPLOYEES(
    ID SERIAL PRIMARY KEY,
    NAME VARCHAR(50)
);

```

Figure 4.15 – Chat Table Creation Script



```

CREATE TABLE SESSIONS(
    ID SERIAL PRIMARY KEY,
    USER_ID INT REFERENCES USERS(ID),
    EMPLOYEE_ID INT REFERENCES EMPLOYEES(ID),
    IsCLOSED BOOLEAN,
    timeStarted timestamp
);


CREATE TABLE MESSAGES(
    ID SERIAL PRIMARY KEY,
    SESSION_ID INT REFERENCES SESSIONS(ID),
    M_TEXT TEXT,
    DATE_SENT TIMESTAMP,
    IS_EMP BOOLEAN
);

```

Figure 4.16 - Chat Table Creation Script

Table Users – used for storing the customer information like name, which is stored in column Name, company which is stored in the Company column, phone number, which is stored in the Telnum column and email, which is stored in the Email column. The id column automatically generated and is a primary key (A primary key is a special relational database table column (or combination of columns) designated to uniquely identify all table records) (Technopedia Inc. - What is primary key?, 2019), which is used for making the relation with other tables.

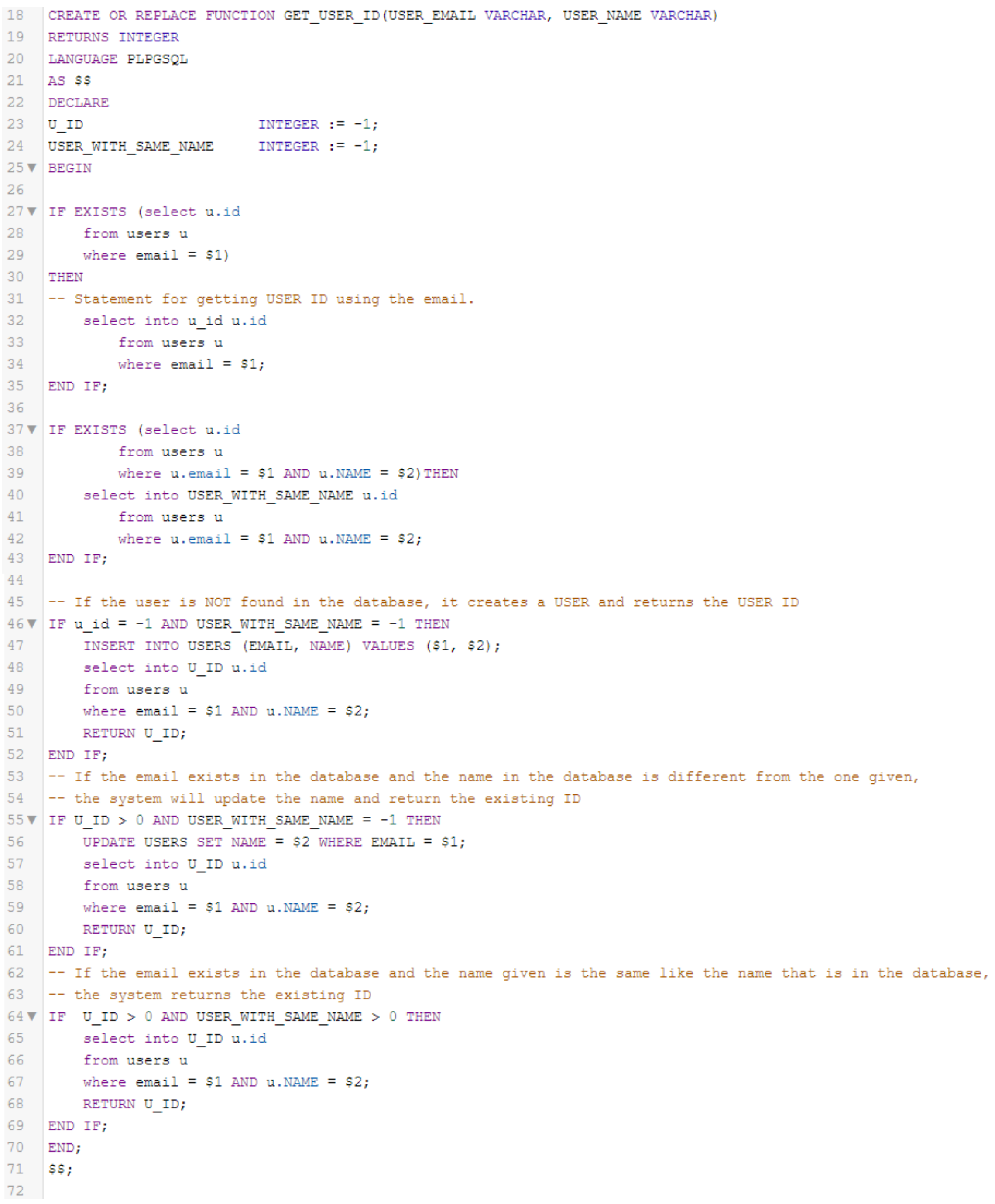
Table Employees – used for storing the HB Products employee information. The information that is stored in this table is the employee name, hold in the column name. The id column is automatically generated and is a primary key (Technopedia Inc. - What is primary key?, 2019), which is used for making the relation with other tables.

Table Sessions – used for saving information about sessions. Each row of the table represents a live chat between a customer and an HB Products employee. This table has a relation with tables Users and Employees. Column Used Id is keeping the id of the customer who initiated the chat session and column Employee Id is holding the id of the employee who took the session. The isClosed column is a Boolean column and it has a value of true if the session was closed or false if the session is still open. TimeStarted is a column, which is keeping the date and time when the session was created.

Table Messages – used for storing the messages data. Each row is a single message that was sent to a chat session. This table has a relation with the Sessions table using the Session Id column. The mText column stores the value of the message and the Date Send column has the date and time that the message was sent. The isEmp column is a Boolean column and it stores a value of true if the message was sent from the employee or a false value if the message was sent from the customer.

4.3.1 Chat Database Functions

Databases are created to deal with data and have better performance when it comes to filtering and checking through data compared to the conventional programming languages. That is why for reducing the work of the Chat API and the requests made to the PostgreSQL server, database functions were created, which take the load for making the inserts in the chat tables and handling the insertions of wrong data. For writing the functions PL/pgSQL was used. This is the procedural language for PostgreSQL. (The PostgreSQL Global Development Group - PL/pgSQL, 2019) PL/pgSQL is a loadable procedural language for the PostgreSQL database system. One of the main functions is shown on Figure 4.17, followed by an in-depth description.



```sql
CREATE OR REPLACE FUNCTION GET_USER_ID(USER_EMAIL VARCHAR, USER_NAME VARCHAR)
RETURNS INTEGER
LANGUAGE PLPGSQL
AS $$
DECLARE
U_ID                    INTEGER := -1;
USER_WITH_SAME_NAME     INTEGER := -1;
BEGIN

IF EXISTS (select u.id
    from users u
    where email = $1)
THEN
-- Statement for getting USER ID using the email.
    select into u_id u.id
        from users u
        where email = $1;
END IF;

IF EXISTS (select u.id
        from users u
        where u.email = $1 AND u.NAME = $2)THEN
    select into USER_WITH_SAME_NAME u.id
        from users u
        where u.email = $1 AND u.NAME = $2;
END IF;

-- If the user is NOT found in the database, it creates a USER and returns the USER ID
IF u_id = -1 AND USER_WITH_SAME_NAME = -1 THEN
    INSERT INTO USERS (EMAIL, NAME) VALUES ($1, $2);
    select into U_ID u.id
    from users u
    where email = $1 AND u.NAME = $2;
    RETURN U_ID;
END IF;
-- If the email exists in the database and the name in the database is different from the one given,
-- the system will update the name and return the existing ID
IF U_ID > 0 AND USER_WITH_SAME_NAME = -1 THEN
    UPDATE USERS SET NAME = $2 WHERE EMAIL = $1;
    select into U_ID u.id
    from users u
    where email = $1 AND u.NAME = $2;
    RETURN U_ID;
END IF;
-- If the email exists in the database and the name given is the same like the name that is in the database,
-- the system returns the existing ID
IF  U_ID > 0 AND USER_WITH_SAME_NAME > 0 THEN
    select into U_ID u.id
    from users u
    where email = $1 AND u.NAME = $2;
    RETURN U_ID;
END IF;
END;
$$;
```

Figure 4.17 – PL/pgSQL Function

Figure 4.17 shows GET_USER_ID function, which is used for getting a user id when creating the chat session. To begin with, this function is taking two parameters USER_EMAIL, the email of the user, and USER_NAME, the name of the user and it is returning an integer (number) - the id that the database has assigned to the user. In the Declare of the function, two variables are made U_ID and USER_WITH_SAME_NAME, both have a value of -1 in the beginning. Line 27 to 35 is an IF EXIST statement, which is checking if a user with the given email exists in the database and if the select statement returns a result, the system assigns it to U_ID. From lines 37 to 43 is a second IF EXIST, that is checking if there is a user with the same email and name in the database, and if there is, it assigns the id to USER_WITH_SAME_NAME. The last step is to check the values of the two variables using the three IF statements that start from line 46 and end on line 69. The first IF is checking if both variables are -1, which means that there is no user with this email and name in the system, then the database is going to make an insert in the database saving the customer name and email and will return the id that was generated. The second IF is checking if there is a user with the given email but not with the same name. In this case, the system will update the customer name in the database, and it will return the id that was assigned to the customer. The last IF statement is checking if there is a customer with the same email and name as the one given as parameters and if this is true it will return the user id without making any insert or changes to the data.

IMPORTANT NOTE: The full source code could be found in Appendix H

5 Test

The purpose of the test section is to document the result of the testing performed during the project period; to verify if the content of the requirements section has been fulfilled; how is the system tested; which testing strategies were used e.g. White Box (Unit Test), Black Box, etc.
The project started with the intentions of following the Test Driven Development(TDD) software process with making the test cases first. For the implementation phase the development team wanted to automatize the testing process by implementing numerous unit tests. It was discovered that most of the features cannot be tested by unit testing (stuff popping out on the screen, data binding lists – mainly UI elements). For this exact reason, the focus of testing was mainly in system testing where the development team tried to tackle the system with random inputs.
The most crucial parts from the applications – the classes responsible for making requests and gathering data from the API were developed in their own classes so that they are easily testable. To ensure that the system is going to work as intended, TDD (Test Driven Development) guided by ZOMBIES testing (zero, one, many, boundary behaviors, interface definition, exercise exceptional behavior, simple scenarios, simple solutions) was used.

To defend the thesis that the list of functional requirements was implemented, four types of tests were performed:

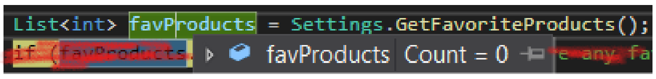
- **Unit testing** – This type of testing is performed to ensure that individual units of the system perform as intended.
- **Integration Testing** – This type of testing is performed to ensure that small subsystems work as intended.
- **System testing** – This type of testing is performed with all units and subsystems linked to a working system.
- **Acceptance testing** – This type of testing is performed by the product owner.

5.1 Unit test

Unit testing is the level of software testing where individual units are tested. The purpose of unit tests is to ensure that each individual unit performs as intended. A unit is the smallest component of a system, usually has one or two input fields and a single output. For this project, the unit tests were not automated but performed manually.

5.1.1 Adding products to favorites

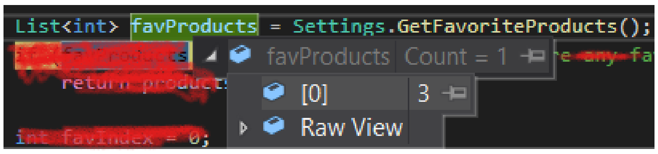
To perform the testing for the adding products to favorites function the following sequence of actions was performed:

1. Go to the Products tab and wait for it to load the product list.
The debugger was attached to a breakpoint in the sorting method which puts the favorite products on top of the list. The list of ID's for the favorite products was as it follows: 

```
List<int> favProducts = Settings.GetFavoriteProducts();
[redacted]  favProducts  Count = 0  [redacted]
```

Figure 5.1- Favorite List before Adding a Product

2. After that, the third product from the list was chosen and in the Product Page, the "Mark as Favourite" button was clicked.

3. After closing the pop-up message coming after adding the product in favorites the application was navigated back to the products tab. The products tab was then refreshed so that the list gets resorted. The list of ID's for the favorite products was as it follows: 

```
List<int> favProducts = Settings.GetFavoriteProducts();
[redacted]  favProducts  Count = 1  [redacted]
              [0]         3
              Raw View
```

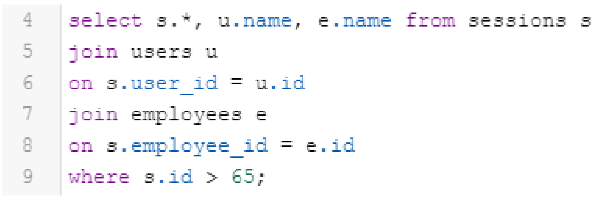
Figure 5.2 – Favorite List After Adding a Products

As can be seen in the screenshot the product count is now one and the ID of the favorite product at index 0 in the list is equal to 3.

NOTE: The tests were performed in the debugger because the data is saved on the device's local storage which the unit tester cannot read!

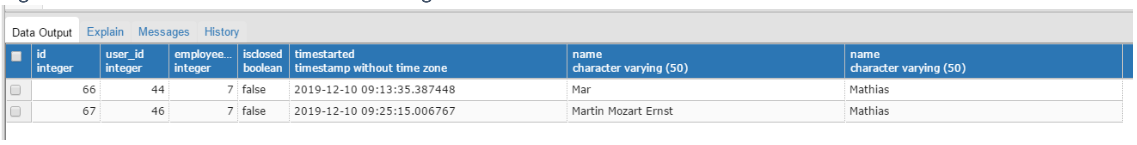
5.1.2 Opening new session

For testing the new_session function, the following steps were made.



```
select s.*, u.name, e.name from sessions s
join users u
on s.user_id = u.id
join employees e
on s.employee_id = e.id
where s.id > 65;
```

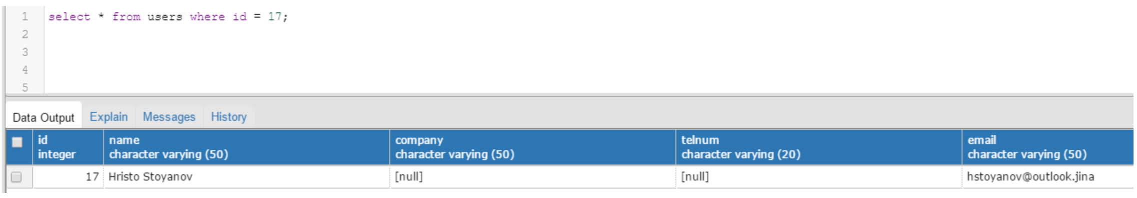
Figure 5.3 – Select Statement for Getting Chat Sessions



id integer	user_id integer	employee... integer	isclosed boolean	timestarted timestamp without time zone	name character varying (50)	name character varying (50)
66	44	7	false	2019-12-10 09:13:35.387448	Mar	Mathias
67	46	7	false	2019-12-10 09:25:15.006767	Martin Mozart Ernst	Mathias

Figure 5.4 – Result of the Select Statement

1. The list of all sessions was shown by executing the statement in figure 5.3. The result from that can be seen in figure 5.4.

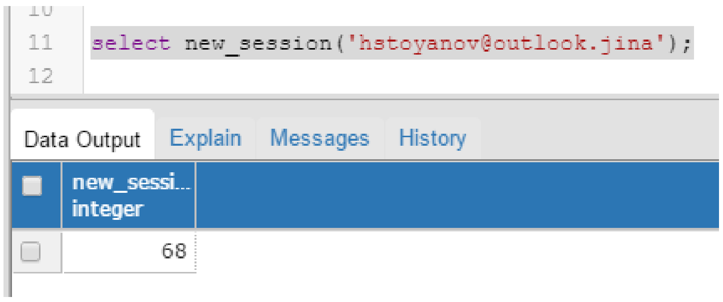


```
select * from users where id = 17;
```

id integer	name character varying (50)	company character varying (50)	telnum character varying (20)	email character varying (50)
17	Hristo Stoyanov	[null]	[null]	hstoyanov@outlook.jina

Figure 5.5 – User with ID: 17

2. After getting the list of the current sessions, a user email was needed for the function parameter and the way for getting one was by running the SQL from figure 5.5, which gives the result of a user with id: "17", name: "Hristo Stoyanov", company: "Null", telnum: "Null" and email: "hstoyanov@outlook.jina".

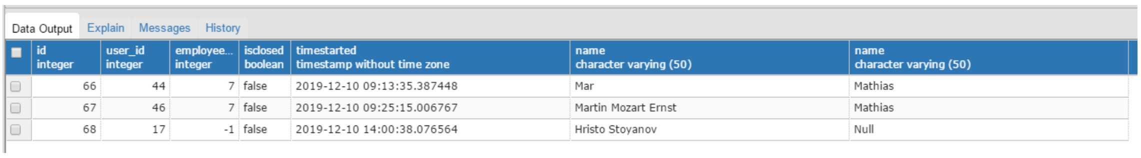


```
select new_session('hstoyanov@outlook.jina');
```

new_sessi... integer
68

Figure 5.6 – Result of Calling a Function

3. To call the function the SQL from figure 5.6 was executed. The function has a return type “INTEGER”, which is the id of the session. The function returns 68 when run.



id integer	user_id integer	employee... integer	isclosed boolean	timestarted timestamp without time zone	name character varying (50)	name character varying (50)
66	44	7	false	2019-12-10 09:13:35.387448	Mar	Mathias
67	46	7	false	2019-12-10 09:25:15.006767	Martin Mozart Ernst	Mathias
68	17	-1	false	2019-12-10 14:00:38.076564	Hristo Stoyanov	Null

Figure 5.7 – Result of Select Statement

4. On figure 5.7 you can see the result of running the select statement from figure 5.3, which is showing the list of sessions after running the function SQL.

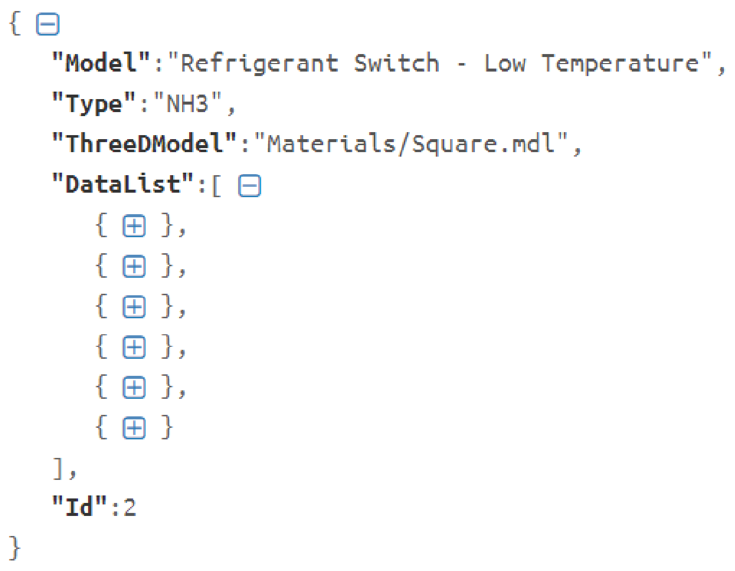
It can be seen that the function is doing the expected action, which is inserting a new session, assigning the user id to it, putting -1 for employee id and returning the id of the session that was created.

5.2 Integration testing

The integration testing was done by developing small subsystems into prototype applications and then integrating them into the main application. An example of that would be the chat subsystem. The chat subsystem was initially developed on a separate application and then transferred into the main app as a button in the “Contact Us” tab.
The integration testing is a type of testing performed to ensure that a composition of units works together as expected. This includes connections between different architectural layers. The examples mentioned in this section go through all of the layers (from application to data layer).

5.2.1 Product manager tests

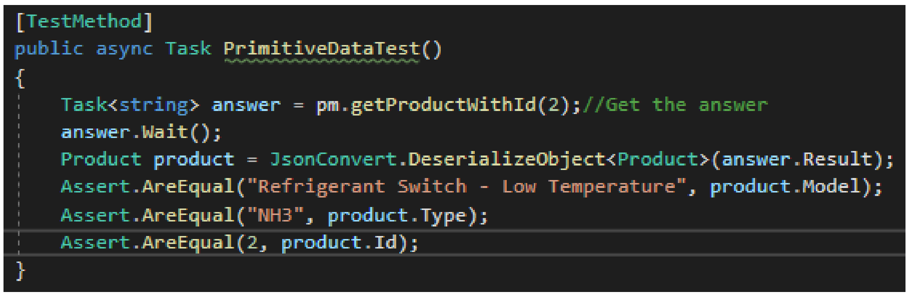
The first method that has been tested is the “getProductWithID(int id)” method. This will be a testament if the API can handle all calls and if the product manager can handle all exceptions also.
In the database, we have a product with the given variables:



```
{ ⊟
   "Model":"Refrigerant Switch - Low Temperature",
   "Type":"NH3",
   "ThreeDModel":"Materials/Square.mdl",
   "DataList":[ ⊟
      { ⊞ },
      { ⊞ },
      { ⊞ },
      { ⊞ },
      { ⊞ },
      { ⊞ }
   ],
   "Id":2
}
```

Figure 5.8 – API Response on Products Request

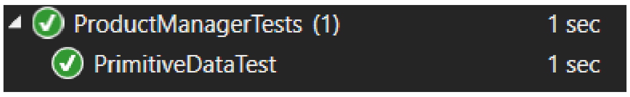
Let's start with the simple stuff and check the primitive data type variables – Model, Type, ThreeDModel, Id. As seen in the picture below a request is made by the product manager and a Product object is made. After that 3 assertations are made.



```
[TestMethod]
public async Task PrimitiveDataTest()
{
    Task<string> answer = pm.getProductWithId(2);//Get the answer
    answer.Wait();
    Product product = JsonConvert.DeserializeObject<Product>(answer.Result);
    Assert.AreEqual("Refrigerant Switch - Low Temperature", product.Model);
    Assert.AreEqual("NH3", product.Type);
    Assert.AreEqual(2, product.Id);
}
```

Figure 5.9 – Primitive Data Test

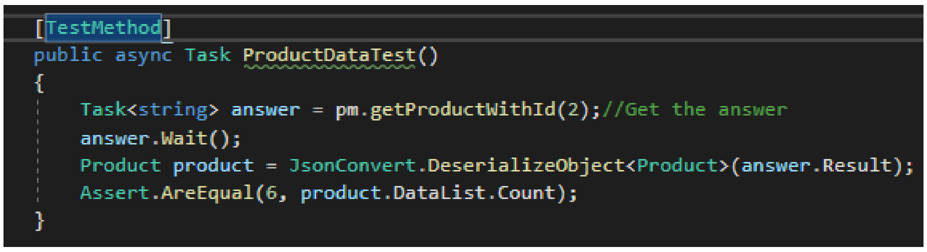
And these are the results after the test has been run:



```
◢ ✓ ProductManagerTests (1)      1 sec
    ✓ PrimitiveDataTest          1 sec
```

Figure 5.10 – Primitive Data Test Result

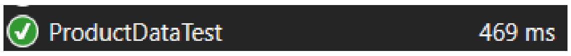
After checking the primitive types, it is good to check if the ProductData list retrieved all product data objects from the database. As seen in the image from the database above, there are 6 product data objects. The following test case was made.



```
[TestMethod]
public async Task ProductDataTest()
{
    Task<string> answer = pm.getProductWithId(2);//Get the answer
    answer.Wait();
    Product product = JsonConvert.DeserializeObject<Product>(answer.Result);
    Assert.AreEqual(6, product.DataList.Count);
}
```

Figure 5.11 – Product Data Test

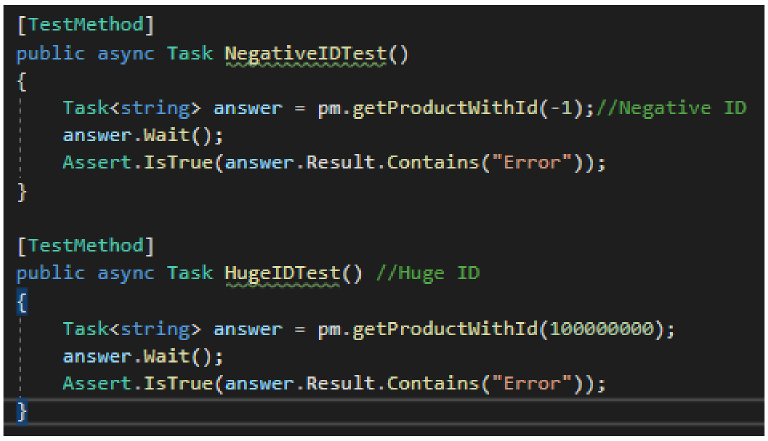
This is the result after running the test case:



```
✓ ProductDataTest          469 ms
```

Figure 5.12 - Product Data Test Result

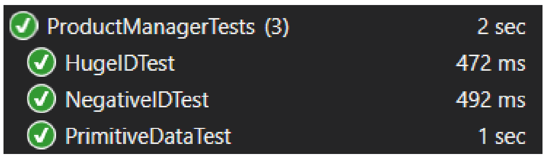
Now that it is proven that the "good cases work", the exceptions should be checked. The way that the ProductManger class is implemented, if an error is encountered it should return a string starting "Error: X" where X is the exception encountered. To check that the exceptions are handled two more tests were made. One testing passing a negative integer as a parameter, and one passing a huge integer as a parameter.



```
[TestMethod]
public async Task NegativeIDTest()
{
    Task<string> answer = pm.getProductWithId(-1);//Negative ID
    answer.Wait();
    Assert.IsTrue(answer.Result.Contains("Error"));
}

[TestMethod]
public async Task HugeIDTest() //Huge ID
{
    Task<string> answer = pm.getProductWithId(100000000);
    answer.Wait();
    Assert.IsTrue(answer.Result.Contains("Error"));
}
```

Figure 5.13 – Negative ID Test and Huge ID Test

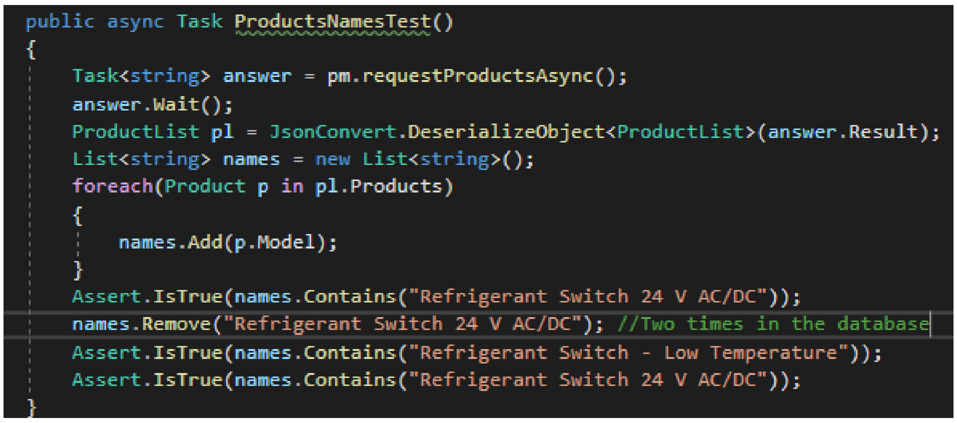
The test results are as follows:



```
✓ ProductManagerTests (3)      2 sec
  ✓ HugeIDTest               472 ms
  ✓ NegativeIDTest           492 ms
  ✓ PrimitiveDataTest          1 sec
```

Figure 5.14 - Negative ID Test and Huge ID Test Results

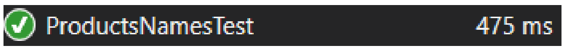
The other method that the ProductManager has is the "getProductsAsync()" method. It retrieves products with their Name and Type properties, as well as their Thumbnail image links. The following test was performed to confirm that the method works as intended. The test checks if the product list contains products with the names of the products added in the database. The following test was executed:



```
public async Task ProductsNamesTest()
{
    Task<string> answer = pm.requestProductsAsync();
    answer.Wait();
    ProductList pl = JsonConvert.DeserializeObject<ProductList>(answer.Result);
    List<string> names = new List<string>();
    foreach(Product p in pl.Products)
    {
        names.Add(p.Model);
    }
    Assert.IsTrue(names.Contains("Refrigerant Switch 24 V AC/DC"));
    names.Remove("Refrigerant Switch 24 V AC/DC"); //Two times in the database
    Assert.IsTrue(names.Contains("Refrigerant Switch - Low Temperature"));
    Assert.IsTrue(names.Contains("Refrigerant Switch 24 V AC/DC"));
}
```

Figure 5.15 – Product Names Test

The test result is as it follows:



```
✓ ProductsNamesTest          475 ms
```

Figure 5.16 - Product Names Test Result

To conclude the tests for the ProductManager the following test was executed. The test was run with the test device disconnected from internet services so there was no connection to the online API.

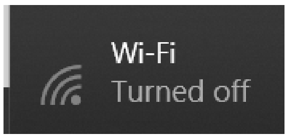
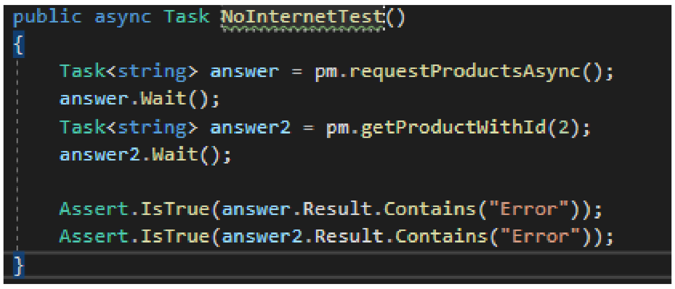


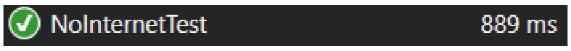
Figure 5.17 – Figure Shows that Wi-Fi is Turned OFF



```
public async Task NoInternetTest()
{
    Task<string> answer = pm.requestProductsAsync();
    answer.Wait();
    Task<string> answer2 = pm.getProductWithId(2);
    answer2.Wait();

    Assert.IsTrue(answer.Result.Contains("Error"));
    Assert.IsTrue(answer2.Result.Contains("Error"));
}
```

Figure 5.18 – No Internet Test

The result is as it follows:

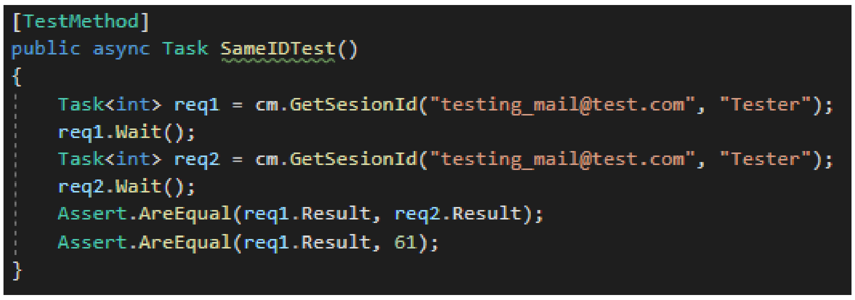


```
NoInternetTest    889 ms
```

Figure 5.19 – No Internet Test Result

5.2.2 Chat manager testing

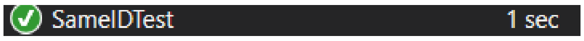
Firstly, the "GetSessionID" method was tested in order to verify that the method returns the same session ID when multiple requests are done from the same user. A session from the user used in the test method has been created prior to running the test. The ID of the session of the user is 61. The following test method was executed:



```
[TestMethod]
public async Task SameIDTest()
{
    Task<int> req1 = cm.GetSesionId("testing_mail@test.com", "Tester");
    req1.Wait();
    Task<int> req2 = cm.GetSesionId("testing_mail@test.com", "Tester");
    req2.Wait();
    Assert.AreEqual(req1.Result, req2.Result);
    Assert.AreEqual(req1.Result, 61);
}
```

Figure 5.20 – Same ID Test

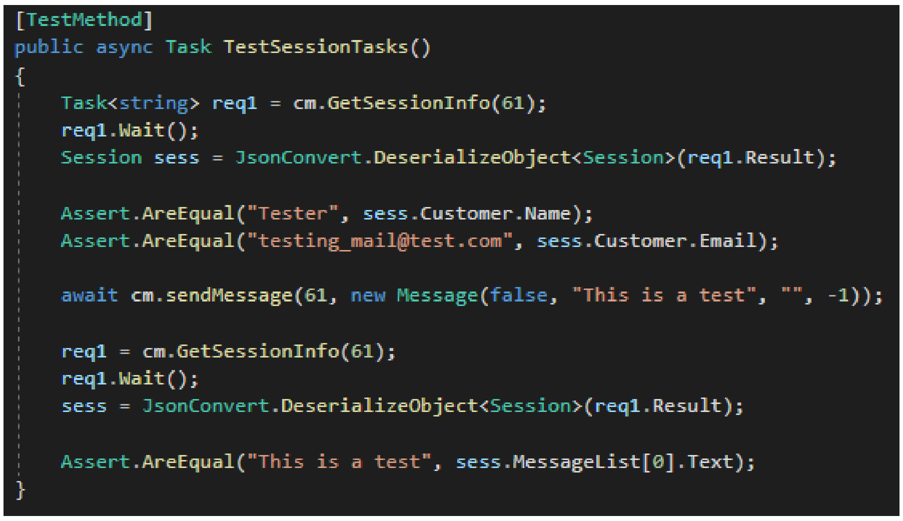
The result of the test is as follows:



```
SameIDTest    1 sec
```

Figure 5.21 – Same ID Test Result

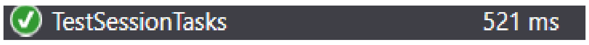
To test the session operations the following method was implemented:



```
[TestMethod]
public async Task TestSessionTasks()
{
    Task<string> req1 = cm.GetSessionInfo(61);
    req1.Wait();
    Session sess = JsonConvert.DeserializeObject<Session>(req1.Result);

    Assert.AreEqual("Tester", sess.Customer.Name);
    Assert.AreEqual("testing_mail@test.com", sess.Customer.Email);

    await cm.sendMessage(61, new Message(false, "This is a test", "", -1));

    req1 = cm.GetSessionInfo(61);
    req1.Wait();
    sess = JsonConvert.DeserializeObject<Session>(req1.Result);

    Assert.AreEqual("This is a test", sess.MessageList[0].Text);
}
```

Figure 5.22 – Session Operation Test

This method tests if the info contained in the session matches the created session from the previous method. Not only the customer credentials are tested, but also the message operations (sending a message). The results of the test execution are as it follows:



```
✓ TestSessionTasks                521 ms
```

Figure 5.23 - Session Operation Test Results

5.3 System Testing

One of the ways for a developer to verify that its product is working, and it is developed according to the customers' requirements is to do a system testing. System testing is a type of testing where a fully integrated system is tested by the developers to asses if the system complies with the specified requirements. (STF - System Testing, 2019)

Usually, for the system test, a Blackbox testing Is used but since all of the developers knew how the system works it was decided to go with a Whitebox testing and predefined Test Cases. All of the tests were performed on both mobile platforms.
The tests performed in this section are based on predefined test cases (Test Driven Development).

5.3.1 Saved Credentials and Data Autofill

Test Case	
Objective	The customer to be able to save his contact information, so that the system is able to autofill it for later use.
Prerequisites	The customer doesn't have his credentials saved into the application.
Test Procedure	The customer does one of the following. Tries to make a product enquiry or tries to send an Email to the company. The customer fills his/her personal data. The data is filled and the "Save my data for further enquiries" is checked. After this is done the "Next" button is pressed. The next time that the customer tries to contact the company by email or make a product enquiry and his/her personal data should be auto filled.
Expected Result	The customer's data is saved, and he/she is able to use the send an email and make an enquiry without refilling the fields.

Actual Result	The actual result matches the expected one
Test Status	**TEST PASSED**

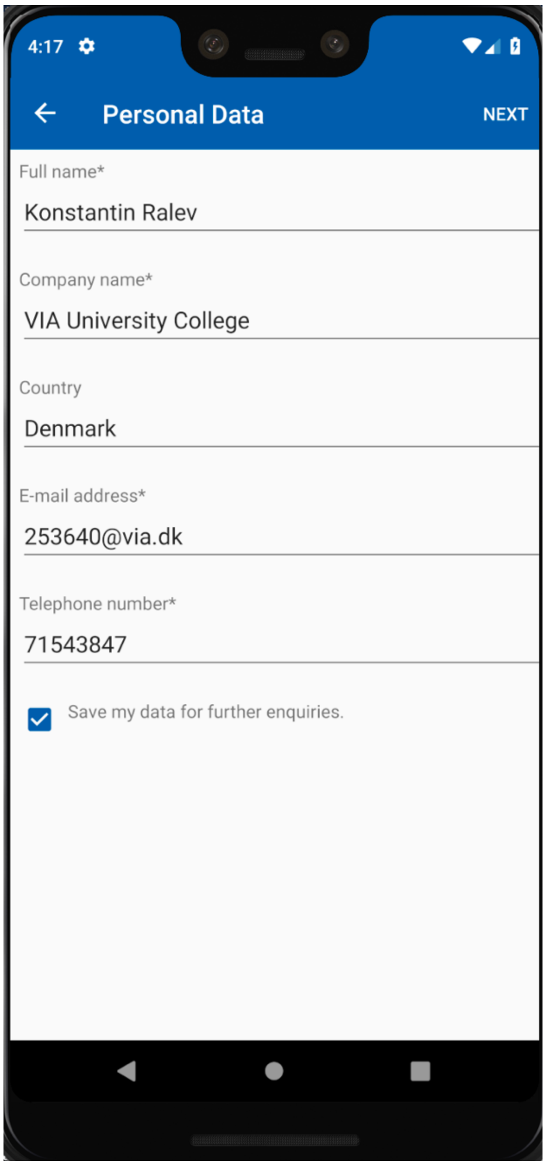

4:17
Personal Data
NEXT
Full name*
Konstantin Ralev
Company name*
VIA University College
Country
Denmark
E-mail address*
253640@via.dk
Telephone number*
71543847
☑ Save my data for further enquiries.

Figure 5.24 - Test Performed on Android

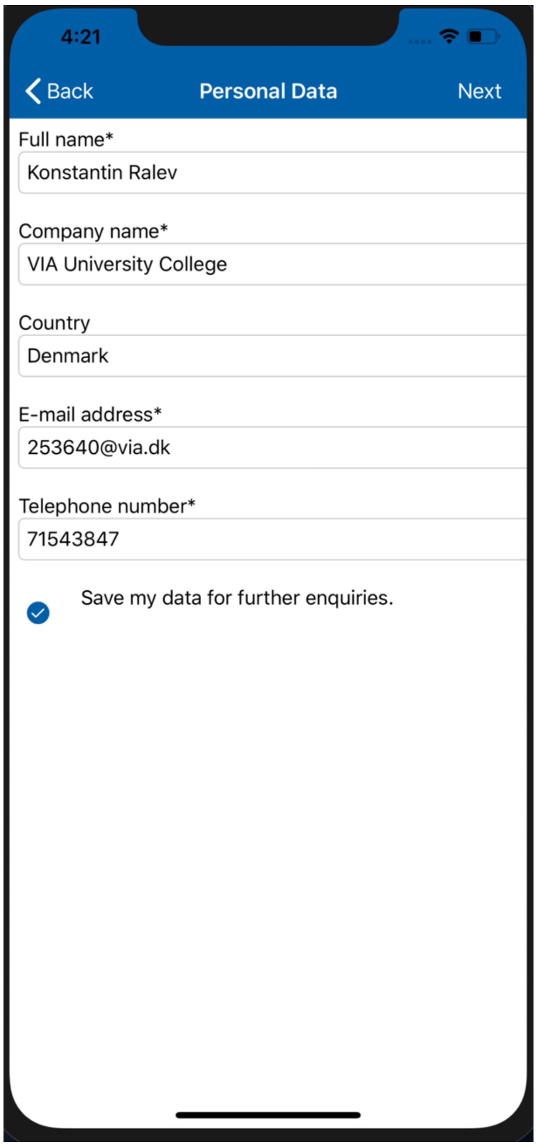
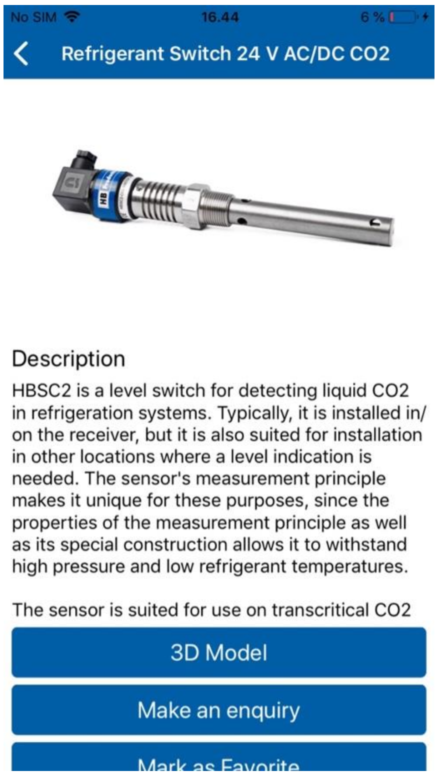

4:21
Back
Personal Data
Next
Full name*
Konstantin Ralev
Company name*
VIA University College
Country
Denmark
E-mail address*
253640@via.dk
Telephone number*
71543847
✓ Save my data for further enquiries.

Figure 5.25 - Test Performed on iOS

5.3.2 The application should have a QR code scanner which identifies a product after scanning its QR code

Test Case	
Objective	The application should have a QR code scanner which identifies a product after scanning its QR code
Prerequisites	The customer has opened the app. Has selected the “Scan QR”. Has granted camera permissions on the corresponding platform. The customer's mobile has internet connectivity.
Test Procedure	A QR code with the number 1 is generated using a QR code generator. The scan button in the “Scan QR” page is pressed. The QR code is placed on the scanner red line and is scanned. **Exception Scenario**: A QR code with a value that is not assigned to a sensor in the database is generated and scanned.
Expected Result	The customer is shown the scanned sensor data, user manuals, 3D model, etc… **Exception Scenario**: The customer is prompted with a pop-up saying “ERROR! Product not recognized. Please try again.”
Actual Result	The actual result matches the expected one
Test Status	**TEST PASSED**

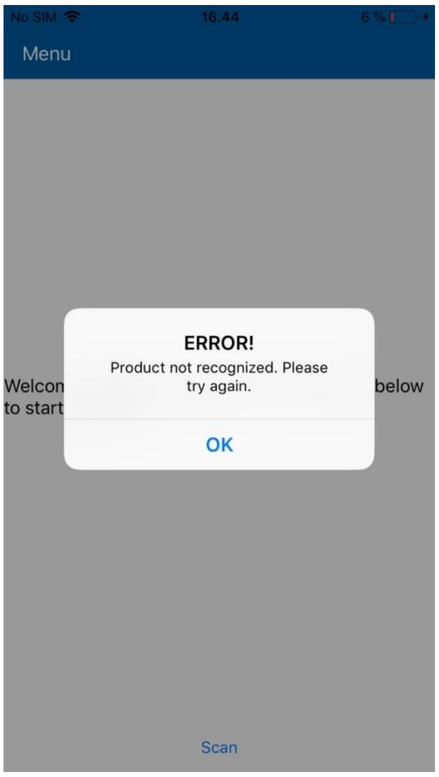

No SIM 16.44 6 %

Refrigerant Switch 24 V AC/DC CO2

Description

HBSC2 is a level switch for detecting liquid CO2 in refrigeration systems. Typically, it is installed in/on the receiver, but it is also suited for installation in other locations where a level indication is needed. The sensor's measurement principle makes it unique for these purposes, since the properties of the measurement principle as well as its special construction allows it to withstand high pressure and low refrigerant temperatures.

The sensor is suited for use on transcritical CO2


No SIM 16.44 6 %

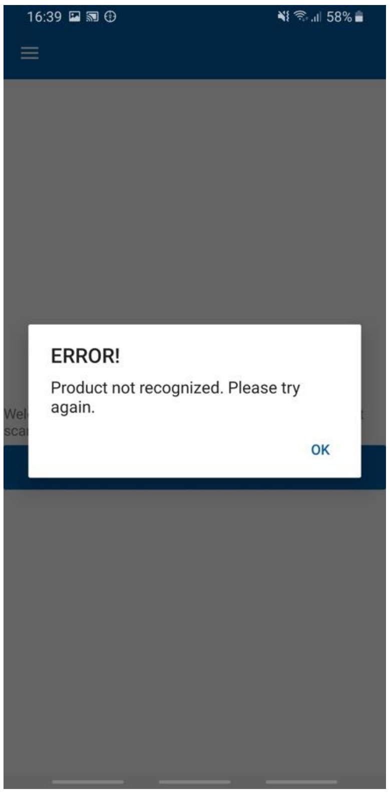
Menu

ERROR!

Product not recognized. Please try again.

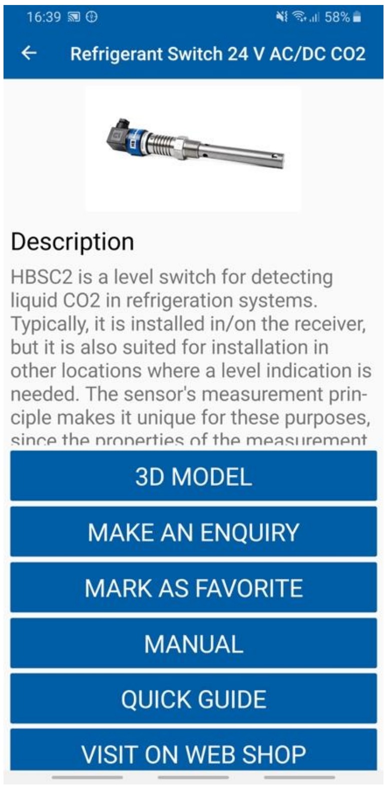
Figure 5.26 and 5.27

Test Performed on iOS


16:39 58%

ERROR!

Product not recognized. Please try again.


16:39 58%

Refrigerant Switch 24 V AC/DC CO2

Description

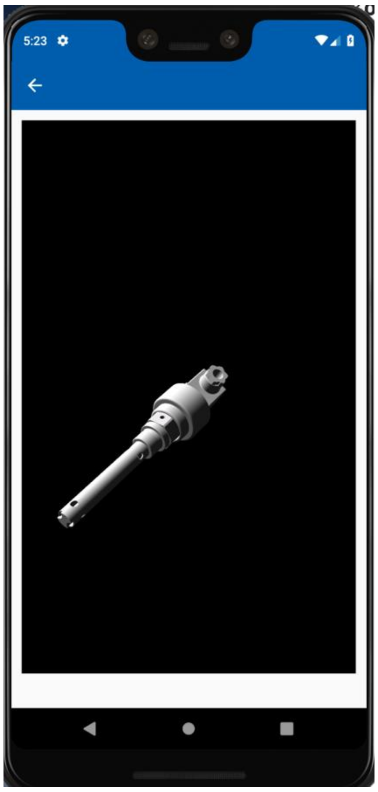
HBSC2 is a level switch for detecting liquid CO2 in refrigeration systems. Typically, it is installed in/on the receiver, but it is also suited for installation in other locations where a level indication is needed. The sensor's measurement principle makes it unique for these purposes, since the properties of the measurement

Figure 5.28 and 5.29

Test Performed on Android

5.3.3 The customer should be able to see a 3D representation of the selected sensor and interreact with it

Test Case	
Objective	The customer should be able to see a 3D representation of the selected sensor and interreact with it
Prerequisites	The customer has opened the app. Has either scanned a sensor’s QR code or selected a sensor from the “Products” tab. The customer’s mobile has internet connectivity.
Test Procedure	The “3D Model” button is pressed. After the customer is transferred to the 3D Model Viewer the customer starts moving his/her finger across the screen expecting the sensor to move. Two-finger “pinch to zoom” gesture is also tested at this point.
Expected Result	The customer is shown the selected/scanned sensor 3D model. The customer is able to interact with the sensor. The sensor should not become too small when un-zooming. The sensor should not become too big when zooming.
Actual Result	The actual result matches the expected one
Test Status	**TEST PASSED**



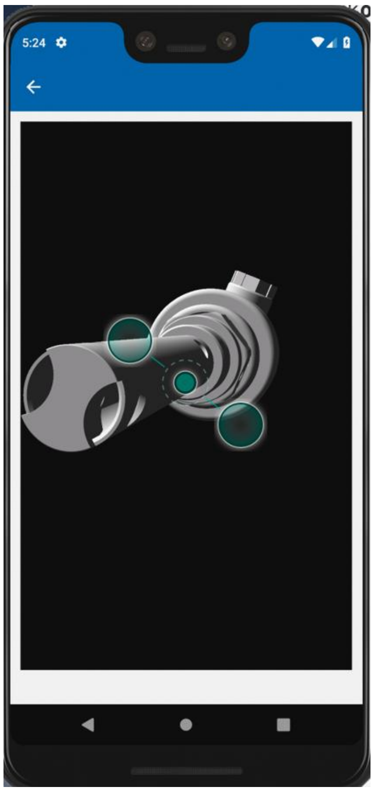
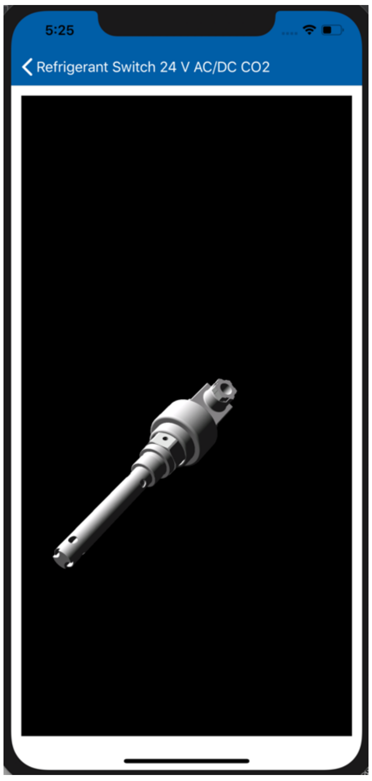


Figure 5.30 and 5.31

Test Performed on Android



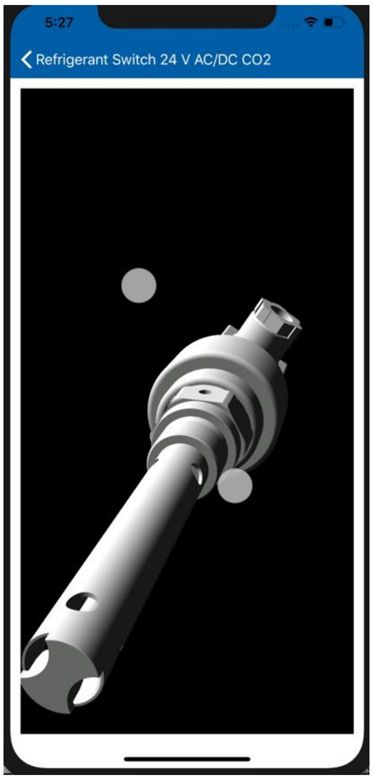
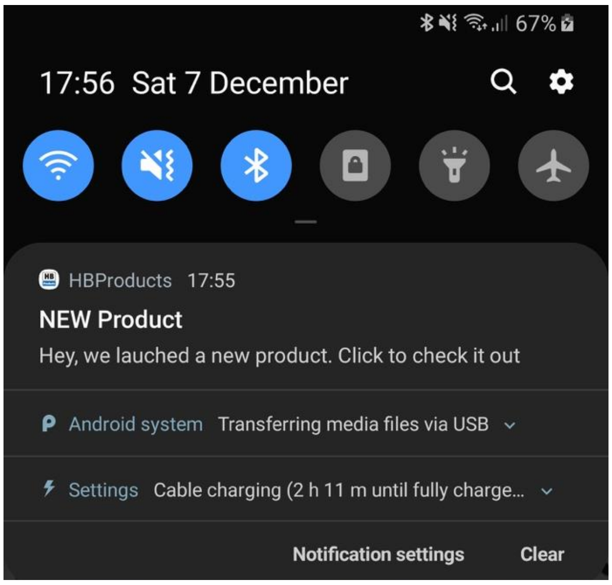


Figure 5.32 and 5.33

Test Performed on iOS

5.3.4 The customer should be able to receive push notifications on both platforms. Both in the foreground and the background.

Test Case	
Objective	The customer should be able to receive push notifications on both platforms. Both in the foreground and the background.
Prerequisites	The customer has opened the app at least once after installation and has granted the app rights to send notifications. The customer’s mobile has internet connectivity.
Test Procedure	A notification using the corresponding platform format (Shown in Figure 4.5 and 4.6) is send using Azure Notification Hub. The customer checks if he/she can receive the notification while using the application. The customer checks if he/she receives the notification while the application is closed.
Expected Result	The customer receives the notification on both platforms, both in the application foreground and background.
Actual Result	The actual result matches the expected one
Test Status	**TEST PASSED**


17:56 Sat 7 December

HBProducts 17:55

NEW Product

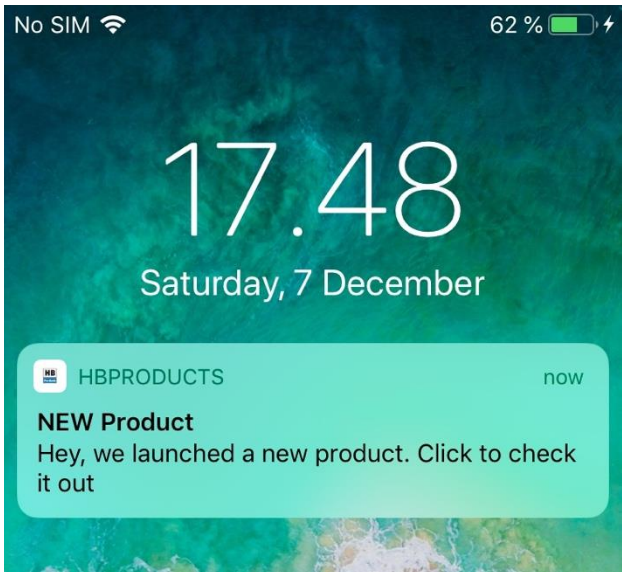
Hey, we lauched a new product. Click to check it out

Android system Transferring media files via USB

Settings Cable charging (2 h 11 m until fully charge...

Notification settings Clear

Figure 5.34 - Test Performed on Android


No SIM 62 %

17.48

Saturday, 7 December

HBPRODUCTS now

NEW Product

Hey, we launched a new product. Click to check it out

Figure 5.35 - Test Performed on iOS

All tests were performed on Android 9 and iOS 12.4

5.4 Acceptance Test

Acceptance testing is done with the product owner at the last stage of the product development to ensure that all the product owner's requirements match the final product. Acceptance test reveals if there are some misunderstandings between the developers' understanding of a requirement and what the product owner meant by defining this requirement. It additionally finds the non-utilitarian issues, for example, exceptions that the developer did not account for while the product owner tests the app in an unconventional manner.

ID	Requirement	Test Status
1	The customer should be able to save his/her contact credentials	**PASS**
2	The application should autofill contact forms using the saved customer credentials.	**PASS**
3	The application should have a sliding menu drawer for navigating.	**PASS**
4	The application should have a QR scanner.	**PASS**
5	The application should be able to recognize the model of a sensor by scanning its QR code.	**PASS**
6	The application should be able to provide the customer with a user manual for a selected product.	**PASS**
7	The application should have an online HB Products webshop.	**PASS**
8	The application should be able to present the customer with a 3D model of the selected sensor.	**PASS**
9	The application should have a Contact us tab.	**PASS**
10	The application should have frequently asked questions tab.	**PASS**
11	The Contact us tab should have "Write an email" and "Chat with us" options.	**PASS**
12	The customer should be able to mark products as favorites.	**PASS**
13	The customer should be able to interact with the 3D model using touch inputs	**PASS**
14	The product owner should be able to send push notifications to the users of the app.	**PASS**
15	The application should have a chat functionality used for customer support.	**PASS**
16	The customer should be able to save the chat session for reviewing later.	**PASS**
17	The customer should be able to make an enquiry for a specific product.	**PASS**
18	The application should prompt the customer for permissions when they are required.	**PASS**

The final product passed the acceptance test successfully. After a discussion with the product owner, it was concluded that the project goal was achieved, and the product owner was happy with the final prototype.

6 Results and Discussion

Based on the system and acceptance testing the project was a success. The development team was able to implement all requirements set in the analysis phase.

The main application offers a clean and simple UI design developed using the Gestalt Principles. Furthermore, it has a QR code scanner that can scan a QR code and show you information about the corresponding product. For proving that this concept works, it is currently connected to the product id in the database. This was done like that because until the end of the project the stakeholders were not certain of the final format of the QR codes that will be present on the actual products.

On the final version of the prototype, there are some limitations for the Push Notification service, the Email Service, the Azure Database and App Services that the application is using. Those limitations are caused because the team uses either a free version of the service or a free trial which at some point will stop – during the development period, free trial of the Microsoft Azure services was used. Thus, some features of the application will also stop working. The aforementioned was implemented like that, considering this project is only a proof of concept and the team only needed those servers until the acceptance test was done and its conclusion was successful.
All system tests were carried out on iOS 12.4 which is the last version that the iPhone 6 provided by VIA University College supports. All tests were successful. When the acceptance test was performed the stakeholder wanted to test the application on his personal phone and the application worked as expected, except that the notifications were not being received on iOS 13. Even though this happened the product owner was happy with the prototype and agreed that all requirements were met.

After the last meeting and the approval, the team wanted to understand why the notifications did not appear on iOS 13 and did some research that presented the team with what was wrong. It looked like Apple just changed how the notification was delivered on the latest iOS 13 which was issued during the project period and unsupported by the test device. The team gained knowledge of how to implement the new notification format but was unable to due to time restrictions and the lack of a test device supporting the latest iOS.

7 Conclusion

At the beginning of the project period, the group was not very confident of how the project will be carried out and if the set goal was realistic. Week by week the development team was researching the different tasks that each member was assigned and was overcoming the different challenges. After the Analysis phase finished and the project’s requirements were finalized and approved by the company the group felt a bit better knowing more about the project.
The team was able to design the application in a way so that it follows design concepts and patterns learned in the previous semesters. All project parts were developed following the Single Responsibility Principle that means that if a part of the project fails the rest of the application still continues to function.
The application was implemented using C# as the main programing language and Xamarin.Forms which is a cross-platform toolkit that targets mobile applications. The project uses 3-tier architecture. It is separated into 3-tiers which are Presentation, Logic, and Business. The project’s presentation layer consists of the application’s User Interface and it is what the customer interacts with. The logic layer is the backend of the application that combines all the managers that handle the products, the chat, the 3d model viewer, and the Web API. The Data Layer of the project was a PostgreSQL Database that was deployed on the Microsoft Azure server.
Overall, the project was a success. All test results matched the expected ones and all the requirements were met. This is not going to be the project’s final state. The product

owner assured the development team that the project will be further developed and released as a real product. The product owner said that the company needs an application/software tool like this so that it could increase the company’s sales and improve customer service.

8 Project future

Generally speaking, the created application was a success, achieved the project’s goal and exceeded the product owner’s expectations. This project had a strict time frame and deadline. That’s why the development team and the project owner set strict requirements and prioritized them so that the final project accomplished everybody’s expectations.
The time frame limited the development team in some parts of the project, for example, the UI. A few times during the project’s development the product owner said that the UI could be improved by adding “more fancy buttons” and this would probably be one of the features that the developers could have implemented if there was more time.
After the project passed its acceptance testing the development team and the product owner discussed the project’s future, what features are missing and could be implemented. The product owner’s propositions were that the Products list and the FAQ could use a search functionality that would allow the user to browse through the products/FAQ faster and find a specific one quicker. This could be easily implemented with the addition of a few UI elements and a simple filter function. One of the product owner’s comments was that currently, the chat functionality of the application lacks notifications when there are new messages in the chat session.
The product owner wanted when a customer starts a chat session, the session to be saved in the background and the customer to be able to continue to browse the application. As soon as the customer receives an answer from the company the customer should be notified. The development team has an idea of how to implement the chat services running in the background so that the customer receives notifications for new incoming messages while the chat is not currently opened. This can be done

by putting –the chat services on a separate background service which keeps running even when the chat page is not open.

That was the only feature that the application missed according to the product owner and overall the project had a favorable outcome.

9 References

College, V. U., 2019. *VIA Software Engineering Project Report Template,* s.l.: VIA University College.

CrossGeeks, 2019. *Azure Push Notification Plugin for Xamarin iOS and Android.* [Online]
Available at: https://github.com/CrossGeeks/AzurePushNotificationPlugin

Dunn, C., 2017. *Introduction to UrhoSharp.* [Online]
Available at: https://docs.microsoft.com/en-us/xamarin/graphics-games/urhosharp/introduction
[Accessed 16 December 2019].

Dunn, C., Britch, D., Schonning, N. & Osborne, J., 2017. *The Model-View-ViewModel Pattern.* [Online]
Available at: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

E. B., 2017. *Bring 3D Models To Life in Augmented Reality with UrhoSharp | Xamarin Blog.* [Online]
Available at: https://devblogs.microsoft.com/xamarin/bring-3d-models-life-augmented-reality-urhosharp/
[Accessed 2019].

Gambill, D., 2017. *Accessing PostgreSql from C sharp.* [Online]
Available at: https://github.com/dalegambill/PostgreSql_and_Csharp

Guru99, 2019. *What is Normalization?.* [Online]
Available at: https://www.guru99.com/database-normalization.html

HB Products A/S, 2019. *HB Products - refrigeration industry sensor specialist.* [Online]
Available at: https:/www.hbproducts.dk/en/hb-products-link/about-us.html
IAmTimCorey, n.d. *Intro to WebAPI - One of the most powerful project types in C#.* [Online]
Available at: https://youtu.be/vN9NRqv7xmY
Interreg - North Sea Region Programme, 2014-2020. *About the GrowIn 4.0 project.* [Online]
Available at: https://northsearegion.eu/growin4/about-the-growin-40-project/
Jinfonet Software, 2019. *3-Tier Architecture: A Complete Overview.* [Online]
Available at: https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/
MasterGruppeG, 2012. *Interaction Design.* [Online]
Available at: http://mastergruppeg.blogspot.com/2012/02/interaction-design.html
Microsoft - What is Xamarin?, 2019. [Online]
Available at: https://dotnet.microsoft.com/learn/xamarin/what-is-xamarin
Microsoft, Architecture - Xamarin, 2017. [Online]
Available at: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/architecture
Phillpotts, K., 2018. *GitHub FormsChat.* [Online]
Available at: https://github.com/kphillpotts/FormsChat
r.-B., 2019. *Urho3D-Blender.* [Online]
Available at: https://github.com/reattiva/Urho3D-Blender
STF - System Testing, 2019. *System Testing.* [Online]
Available at: http://softwaretestingfundamentals.com/system-testing/
Technopedia Inc. - What is primary key?, 2019. *What is primary key?.* [Online]
Available at: https://www.techopedia.com/definition/5547/primary-key
The PostgreSQL Global Development Group - About, 2019. *PostgreSQL: About.* [Online]
Available at: https://www.postgresql.org/about/
The PostgreSQL Global Development Group - PL/pgSQL, 2019. *PL/pgSQL - SQL Procedural Language.* [Online]
Available at: https://www.postgresql.org/docs/7.3/plpgsql.html

T. I. P. L., 2018. *Importance of smartphones in our life.* [Online]
Available at: https://targetstudy.com/articles/importance-of-smartphones-in-our-life.html

Volkerdon, 2018. *MoSCoW prioritization technique.* [Online]
Available at: https://www.volkerdon.com/pages/moscow-prioritisation

Wikipedia - JSON, 2019. *JSON - Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/JSON

Wikipedia - Microsoft Azure, 2019. *Microsoft Azure - Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Microsoft_Azure

Wikipedia - SQL Injection, 2019. *SQL injection.* [Online]
Available at: https://en.wikipedia.org/wiki/SQL_injection

Wikipedia - Web API, 2019. *Web API - Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Web_API

Xamarin, M., 12/4/2018. *NuGet Gallery | UrhoSharp.Forms 1.9.67.* [Online]
Available at: https://www.nuget.org/packages/UrhoSharp.Forms

10 Appendices

- Appendix D – Use Case Descriptions
- Appendix E – System Sequence Diagrams
- Appendix F – Class Diagrams
- Appendix G – Sequence Diagrams
- Appendix H – Source Code
- Appendix I – ER Diagrams
- Appendix J – Activity Diagrams
- Appendix K – Use Case Diagram
- Appendix L – Project Description