Bachelor's thesis

Information and Communications Technology

2022

Eerik Hannula

Developing a mobile application for property maintenance

– Case Pyhä-Luosto Matkailu Oy

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | 38 pages

Eerik Hannula

Developing a mobile application for property maintenance

- Case Pyhä-Luosto Matkailu Oy

The goal of the thesis was to develop a property maintenance mobile application for Pyhä-Luosto Matkailu Oy. The application had to fulfill functionalities and requirements that were determined together with the commissioning company. This thesis discusses the implementation of these requirements, and the basic principles of the development process of a mobile application and server. The communication and exchange of data between these components are also covered.

The project was developed using the .NET development platform, which is a collection of frameworks and tools created by Microsoft. The mobile application was programmed using the Xamarin.Forms framework, and the server used ASP.NET as its core framework. Since these frameworks were part of a single platform, the development process and implementation of features was straightforward.

The result of this thesis was an operational mobile application and server that fulfilled the most essential requirements set for the project. The rest of the functionalities and requirements were planned for further development.

Keywords:

Mobile development, C#, .NET, programming, Xamarin.Forms, ASP.NET

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2022 | 38 sivua

Eerik Hannula

Mobiilisovelluksen kehittäminen kiinteistöhuollon tarpeisiin

- Case Pyhä-Luosto Matkailu Oy

Työn tavoitteena oli kehittää mobiilisovellus kiinteistöhuollon tarpeisiin Pyhä-Luosto Matkailu Oy:lle. Sovellukselle asetettiin vaadittavat ominaisuudet yhdessä Pyhä-Luosto Matkailun kanssa. Tämä opinnäytetyö käsittelee näiden määriteltyjen ominaisuuksien toteuttamista, sekä mobiili- ja palvelinkehityksen keskeisimpiä periaatteita.

Projekti toteutettiin .NET kehitysympäristössä, joka on kokoelma Microsoftin julkaisemia ohjelmistokehityksessä käytettyjä työkaluja. Mobiilisovellus ohjelmoitiin käyttäen Xamarin.Forms ohjelmistokehystä, ja palvelinympäristö toteutettiin ASP.NET ohjelmistokehyksellä. Koska Microsoftin kehittämät työkalut toimivat hyvin toistensa kanssa, työn toteutus oli suoraviivaista.

Työn tuloksena oli toimiva mobiilisovellus, sekä palvelin. Tärkeimmät työlle asetetut tavoitteet täytettiin, ja lopuille tavoitteista laadittiin suunnitelma jatkokehitystä varten.

Asiasanat:

Mobiilikehitys, C#, .NET, ohjelmointi, Xamarin.Forms, ASP.NET

Contents

List of abbreviations

API	Application Programming Interface
DTO	Data Transfer Object
GUI	Graphical User Interface
IDE	Integrated Development Environment
iOS	Mobile Operating System created by Apple Inc.
JSON	Javascript Object Notation
MVC	Model-View-Controller, software design pattern
MVVM	Model-View-ViewModel, software design pattern
REST	Representational State Transfer
XAML	Extensible Application Markup Language

1 Introduction

1.1 Pyhä-Luosto Matkailu Oy

This thesis topic was assigned by Pyhä-Luosto Matkailu Oy [1]. The company specializes in travel and provides accommodation for visiting tourists in Luosto, Lapland. It has been in business for over 20 years.

The customers of Pyhä-Luosto Matkailu can book a cabin for their stay. Once the stay is over and the cabin is empty, the employees of the company will clean and prepare the cabin for the next customer. To keep track of the cleaning jobs, lists are created on paper and distributed to the cleaning personnel. The company wants to shift from these lists to an electronic application and this thesis addresses the development process of the application.

1.2 The purpose of the application

The main purpose of the application is to organize cleaning-related work orders and keep track of the employees' work schedule. Since the locations of the cabins are distributed around Luosto, quick and easy-to-use communication - scheduling system will save time and travel costs for the company's employees. The application automatically saves the users' work hours to a database, from where it is easy to import these hours to the payroll system.

Employees are also able to mark deficiencies in cabin supplies, broken items, and make other notes to the application, where the manager can organize refills and repairs in real time.

1.3 The development environment

This project is carried out by using .NET Software development environment [2], which is a collection of different software development technologies. The .NET

environment was chosen for this project for its comprehensive class library, which provides a wide range of tools for building the required features. Because of this variety of tools, the need for third-party packages and libraries is decreased notably.

This thesis focuses on mobile development with Xamarin.Forms [3], as well as server-side programming with ASP.NET [4].

Xamarin.Forms is a .NET mobile development framework. It will be used to build the mobile application of the project. Xamarin.Forms extends the .NET developer platform with tools and libraries for developing native Android, iOS, macOS and Windows apps, using single shared codebase.

ASP.NET is a .NET framework for producing web applications. It is reliable, fast, easy to use, free and widely known. In this project, ASP.NET will be used to create a server that hosts a web-API for providing the mobile application with data. The API will be connected to SQLite [8] database with Entity Framework [10]. Entity Framework provides an easy way to map objects into database tables. Because of this mapping, fetching and storing data is simple and efficient. The planned architecture of the project can be seen in Figure 1.

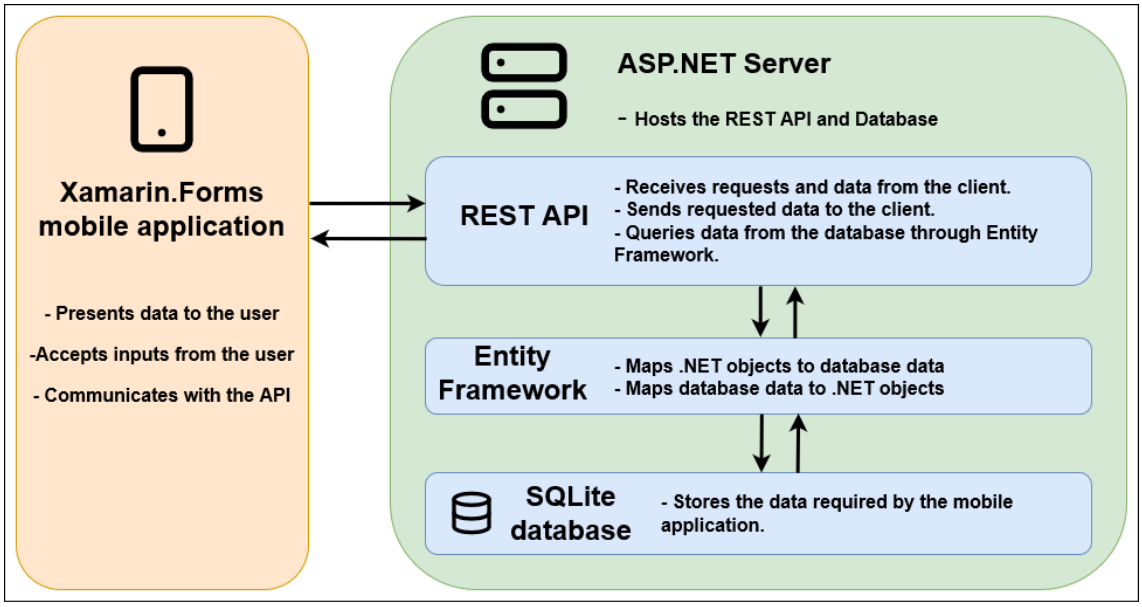

Figure 1. System architecture, the arrows indicate exchange of data between the components.

The thesis is divided into chapters, which depict the logical structure of the development process of a mobile application. In chapter 2, the requirements of the application are covered. Chapter 3 looks into the methods and technologies used in the development, leading to chapter 4, which covers the implementation of these methods. Chapter 5 covers the testing of the application's functionalities, and finally, chapter 6 outlines the results and the plans for further development of this project.

2 Requirements

The requirements for this project are determined by using the MoSCoW-method [11]. This method can be used to sort the requirements into categories, which determine the priority of the requirement.

Requirements labelled as “Must have” are the backbone of the application. They ensure that the application is operational and ready to ship.

“Should have”-label means that the requirement is important, and it should be implemented if possible.

“Could have”-label is for requirements that would be nice to have. They are not essential but implemented if there is time and resources.

Requirements with “Won’t have”-label are not to be implemented. They are still listed, because they might be added later on.

The requirements determined for this project can be seen in Table 1.

Table 1. Prioritized requirements.

Requirement	Priority
The mobile application runs on Android operating system.	Must have
The mobile application can communicate with the server.	Must have
Server stores data used by the mobile application.	Must have
Multiple users can communicate with the server simultaneously.	Must have
User authentication using username and password.	Must have
Users are sorted into two groups: *Organizers* and *Cleaners.*	Must have
Organizer can create and delete *cleaners.*	Must have
Organizer can create and delete cabins, and edit cabin information.	Must have
Organizer can create and edit schedules.	Must have
Cleaner can inspect their work shifts.	Must have
Cleaner can inspect cabins.	Must have
Cleaner can report missing or broken items.	Must have
Cleaner can start and stop their work shifts, timestamps are saved.	Must have
database to store data about users, cabins, schedules and work hours.	Must have
Work hours are automatically calculated.	Should have
State of the cleaning should be visible to all users.	Should have
Time when the cabin is ready for customers should be visible to all users.	Should have
Language support for english and finnish.	Should have
Cabins are visible on a map.	Could have
Cabins that are under cleaning are highlighted on the map.	Could have
Work hours can be imported to excel.	Could have
Application runs on IOS.	Won’t have
Chat functionality between users.	Won’t have

In the following subchapters, some of the requirements are discussed in more detail.

2.1 Support for multiple simultaneous users

Since there is going to be more than one maintenance person working at the same time, the server needs to be able to handle requests arriving simultaneously. This also means that querying data should be efficient enough, that multiple requests receive their response inside an accepted time window.

2.2 User authentication

To ensure that unwanted activity does not take place in the property maintenance system, user authentication is required. Every employee has their own account, which is used to fetch essential data from the server. On top of provided security, user accounts make it possible to grant specific roles and permissions to the users.

2.3 User specification

The mobile application is required to have two different user account types: Organizer and Cleaner. These account types have their own permissions for how they can interact with the application.

The role of the organizer-accounts in this application is to plan and create work schedules for the maintenance personnel. Organizers are able to add, modify and delete data residing in the database. This data consists of cabins (properties), users, schedules and job information, such as list of tasks, time of customers leaving or arriving, personnel assigned to the job and more.

Cleaner-accounts are able to inspect cabins, schedules and job information. They are able to change the status of the cleaning job. This will save a timestamp to the database, which will be later on used to determine work hours. After the job is done, the cleaner will fill a report template of the job. This report contains the tasks done, deficiencies, broken items, lost property and optional extra information.

2.4 Map functionality

One of the “Could have”-labeled requirement is that the cabins are visible on a map. When the employees are presented with jobs, they can easily determine the location of this job with a map functionality.

2.5 For further development

It was determined that because of the tight schedule, the application will not be ported on iOS during this thesis. For the same reason, a chat functionality was left out of the scope of this project. The importance of these requirements will be re-evaluated on a later date, which is for now unknown.

3 Methods and technologies

In this chapter, the most essential technologies utilized in this project are discussed.

3.1 .NET platform

.NET is developer platform for creating different types of applications. It initially launched as .NET Core in 2016. Where as it's predecessor .NET Framework was used to develop applications to Windows, .NET supports also Linux and macOS operating systems.

Fully supported programming languages on the .NET platform are C#, F# and Visual Basic. This project is created with C#.

The .NET platform houses variety of tools and technologies – ready to use – by the developers. The most essential .NET technologies used in this thesis are: Xamarin.Forms, ASP.NET, Entity Framework and the IDE Visual Studio.

3.2 Microsoft Visual Studio

Microsoft Visual Studio [7] is an IDE, launched in 1997 and continuously developed to this day. It is used for software development and provides a wide range of tools to make it easier. This is a .NET developers best friend when writing almost anything in Microsofts development environment.

3.2.1 NuGet Package Manager

NuGet is an integrated package manager in Visual Studio. It makes the installation of additional libraries straight-forward with its built in search function and one click installation. Visual Studio also presents warnings to the user, if the installed packages are not compatible with the version of the project.

3.3 Git

Git is a version control system used to track changes along the development process of projects. Visual Studio has Git integrated to it, which makes committing new code and resolving conflicts effortless.

3.4 Xamarin.Forms

Xamarin Forms is open-source mobile development framework. It was released in 2014 by Xamarin. Later on, when Microsoft acquired Xamarin, Xamarin.Forms was welcomed to the .NET family.

Xamarin.Forms uses a single codebase to develop applications to multiple platforms. Developers have to write the application only once, and the Xamarin.Forms converts the source code to other platforms.

3.4.1 The MVVM pattern

The MVVM architecture [37] divides the application into three main components: Model, View, and ViewModel. This architecture separates the user interface from the business logic. It reduces boilerplate code and makes it easier to test different parts of the application.

The view component presents data to the user. Where normally information presented to the user is located in views code-behind (GUI-code), in MVVM the view binds to exposed properties on the viewmodel. User inputs like button presses are also sent to the viewmodel instead of handling it in the code-behind.

The model is a data access layer of the application. It holds the data and business logic used in the application.

The viewmodel is a bridge between the view and the model. The viewmodel provides properties and commands which the view can bind to, and handles the logic in which way the data is presented to the user.

Because of this light coupling between the components, changing data and presenting it to the user is easy and efficient (Figure 2).

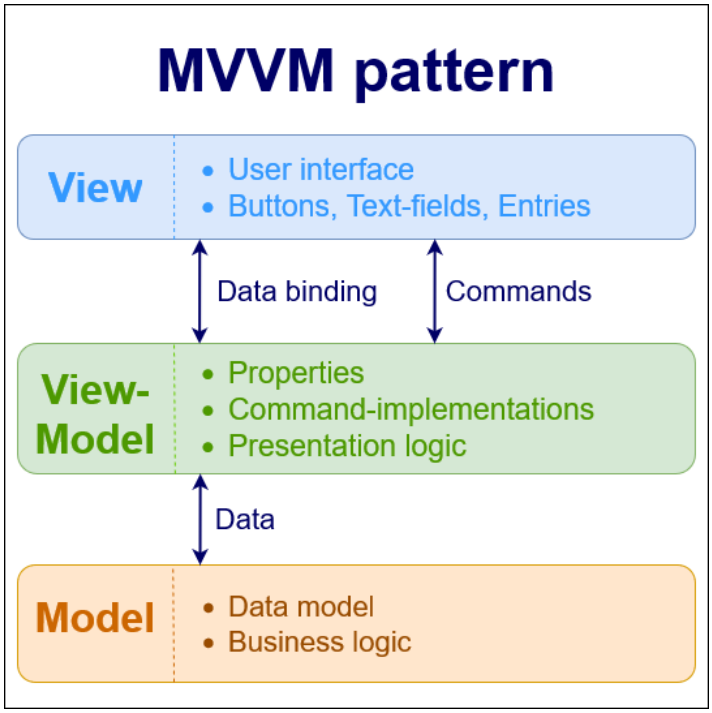

Figure 2. The MVVM reference model.

3.5 ASP.NET Core

ASP.NET Core is a server-side framework created by Microsoft and released in 2016. It is a successor to Microsoft's ASP.NET. It is used for building web applications, IoT applications, services and mobile backends.

In this project ASP.NET Core is used for developing the server responsible for data distribution, storage and the backend of our mobile application.

3.5.1 The MVC pattern

Instead of the MVVM-architecture used in the mobile app, the server is built using MVC-Architecture [6]. The MVC also divides the application into three components: Model, View, and Controller.

The model and the view share the same purpose in MVC, as in MVVM, the model holds data and business logic, and the view presents data to the user.

Controllers are responsible for receiving requests and sending responses to and from the web. It is the mediator between the client and the data that the client wants.

3.5.2 REST API

The server will communicate with the mobile application using REST API. REST stands for representational state transfer. It is a set of architectural constraints and uses HTTP protocol to send and receive requests.

API stands for application programming interface. APIs usually receive requests for data and then send a response accordingly. In this project, the API will be built such that the mobile application sends requests for data residing in the servers database and returns the requested data. The data is carried in the body of a request, or a response and it is encapsulated in JSON-format.

Most commonly used request types in a REST-system are POST, GET, PUT and DELETE. This application also uses a request named PATCH.

GET-request is used when the client requests an object or objects from the server.

POST-request is used when the user requests to upload data on the server.

PUT-request is used when the user requests to update data on the server.

DELETE-request is used when the user requests to delete data from the server.

PATCH-request is similar to PUT-request in that it requests to update data on the server, but instead of updating whole object, the PATCH-request updates only specific properties of an object.

A reference model of a REST API can be seen in Figure 3.

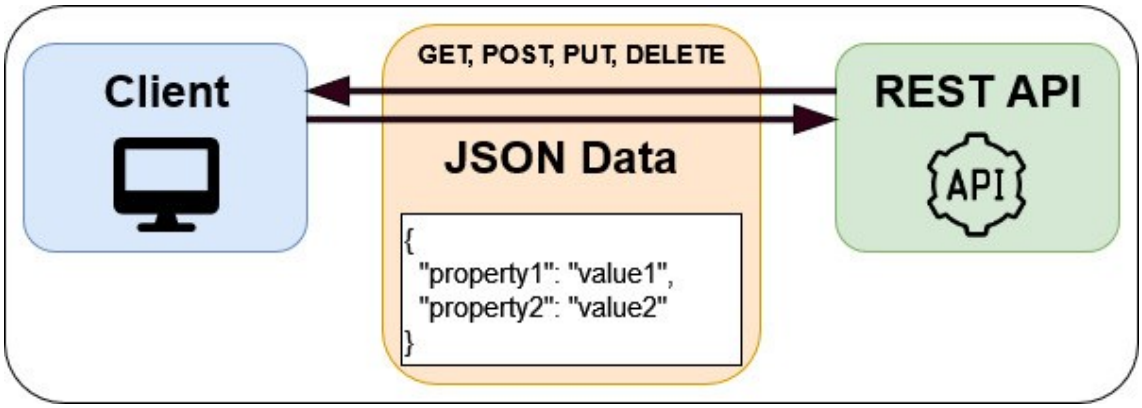

Figure 3. The REST API model.

3.6 NewtonSoft.Json

The data used by the server and the mobile application exists as .NET objects. Before these objects can be sent from the client to the server or vice versa, they need to be converted to JSON-objects. Converting these objects to JSON is tedious task, which is why this project uses a third-party library called “NewtonSoft.Json”. With NewtoSoft.Json, the developer has to write a single line of code, which transforms the .NET objects into JSON-string or JSON-string to .NET objects.

3.7 Entity Framework

When the client requests an object from the server, before responding, the server needs to fetch the corresponding data from the database and build the object that holds the data. This is done by using Entity Framework.

Entity framework is used to map objects into the database. Every object registered to the entity framework has their corresponding table in the database.

If the object has some properties that won't need storing, they can be excluded from the mapping by using “NotMapped” attribute with the corresponding property. Another option is to have separate objects for the database and data transfer.

4 Implementation

This chapter discusses the implementation of the requirements listed in chapter 2, by using the methods and technologies introduced in chapter 3.

4.1 The mobile application

The development process started by creating an empty Xamarin.Forms project in Visual Studio. Folder structure was made keeping the MVVM-pattern in mind: UI components were located in Views-folder, presentation logic in Viewmodels-folder, and data with business logic in Models-folder.

4.1.1 The service layer

In MVVM-pattern, it's sometimes necessary to include a service layer on top of the base components. The service layer implements logic that is outside the scope of viewmodels and models. It also solves the issue with dependency injection, where passing too many object instances through the viewmodel constructor lowers the readability of the code.

The service layer is a container that holds the service-classes used in the application. These services are accessed from viewmodels using service locator, so no dependencies need to be passed when creating viewmodels.

When the application starts, services are registered to the container. For each service, there is an interface which is used to locate the service. The service locator looks for corresponding interface and returns the implementation version of the class.

Services that were created for this project are listed in Table 2.

Table 2. Services implemented in the mobile application.

Service	Description
Navigation service	Handles the logic required to change visible page for the user. Abstraction of the Xamarin.Forms Shell navigation system.
REST service	Logic behind requests that are sent to the server. Receives data from the server and redirects it to waiting classes in the application.
Dialog service	Presents user with pop up messages and confirmation dialogs.
Identity service	Determines if user is logged in. Also determines the user that is logged in. Handles authentication with the REST service.

4.1.2 Switching views

Looking at the requirements of the application, there are lots of different functionalities waiting for implementation. These functionalities will not fit into a single page, which means that switching pages is essential to successful delivery of the application.

In this project, switching pages was done through Xamarin.Forms routing system. Like on a website, routes or paths are used to specify which content will be presented to the user.

In Xamarin.Forms, the route is registered while the application starts. The registration requires two parameters: name of the route, and page to be presented when navigating to this route.

Once the route is registered, it can be used around the application when needed. The actual navigation through pages is done using a navigation service.

4.1.3 Data binding

Data binding is a way to make data visible to the user in an application which uses the MVVM-pattern. Data binding is also used when determining what happens when the user interacts with the elements on the UI-layer.

When a view is about to be presented to the user, it doesn't yet know about the presentation logic residing in a corresponding viewmodel. Before the page is loaded, its binding context needs to be determined. The binding context provides the view a way to access the public properties on a viewmodel.

When the data updates on a viewmodel, the view needs to be informed about it. This was done by implementing the interface "INotifyPropertyChanged" [13]. The interface implements a method "OnPropertyChanged", which invokes an event that notifies the UI about changed information [Figure 4].

```
public event PropertyChangedEventHandler PropertyChanged;

50 references
protected void OnPropertyChanged([CallerMemberName]string propertyName = null)
{
    if (PropertyChanged != null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Figure 4. Implementation of the method responsible for notifying UI, that the data has changed. The method is called from a property setter on a viewmodel, which then notifies the UI to update the binded property on a view.

4.1.4 Authenticating users

To prevent access from unwanted entities, authentication system was one of the projects requirements. Since the application is small-scale and not going to be publicly available, it was determined that simple username – password authentication is enough in terms of security.

A simple system was created for sending the login credentials to the server. The server then compares these credentials with credentials saved to the database. If the credentials match, a response is sent which allows the application to proceed forward from the login page.

The hashing of credentials was not implemented yet. Preferably this always happens before the credentials leave the client application, so that no plain text passwords can be intercepted and read by third parties.

4.1.5 Making the application multilingual

The primary language of the user interface is Finnish, but it was determined that language support for English was also required. To achieve support for multiple languages, text presented to the user must have English and Finnish counterparts.

In .NET environment, text visible to the user can be determined dynamically, meaning while the program is executing. The text strings are saved in resource files, one for English and one for Finnish. When running the application, selected language can be used to determine which resource file should be used, thus presenting the user with correct language.

4.1.6 The user interface

The thesis mainly focuses on the back-end development of a mobile application. However, the user interface is one of the most important parts of a

modern application, which is why this subchapter presents a couple of the most essential pages that the user interacts with. These pages were designed to be used with an android tablet, which has larger screen size than a regular smartphone.

The scheduling page can be seen in Figure 5. This page is used by the organizer to view and create jobs for the employees. A calendar covers the upper part of the page, where the organizer can select a date. The lower part of the page presents a list of jobs planned for the selected date. Each job item in the list shows relevant information about the job. Orange labels indicate the name of the cabin, purple labels indicate job type, pink labels indicate the status of the job and blue labels contain the employees assigned to the job. Under these labels, optional extra information can be observed.

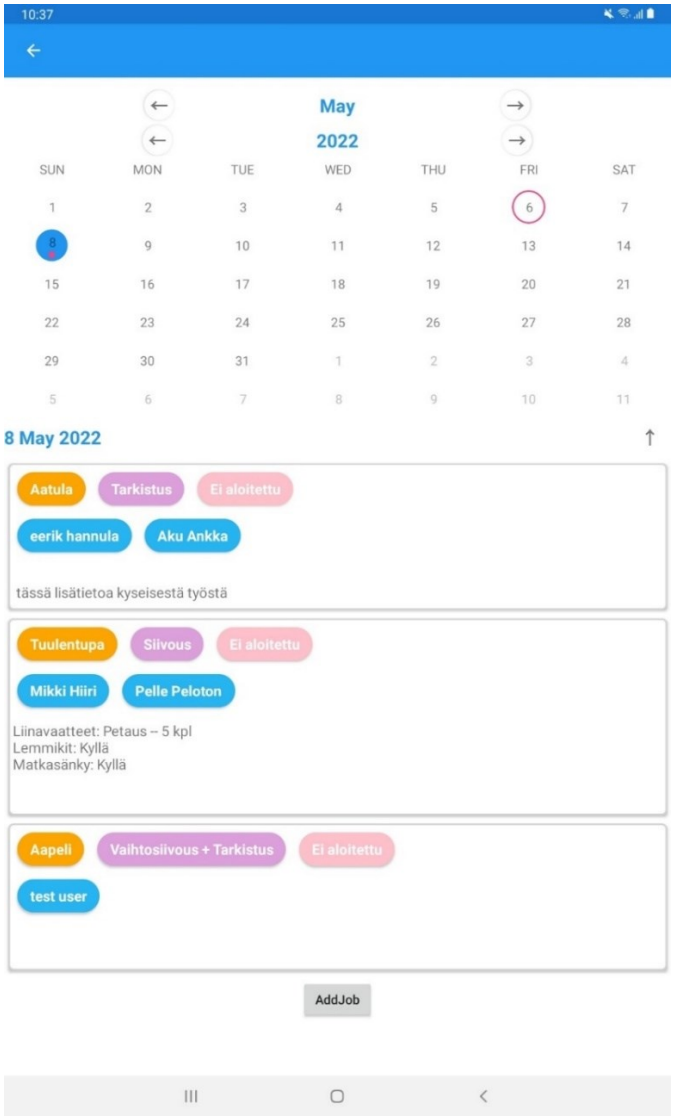

Figure 5. The scheduling page.

The job page can be seen in Figure 6. The maintenance personnel use this page to find instructions about the job and write a report of the cleaning. On top of the page, information about the job is presented. The same label colors have been used as in the scheduling page. Cabin details can be expanded by tapping the arrow pointing down. The cabin details contain information about the property such as heating systems, air conditioning, waste management, snow plowing and so on. This expander also contains instructions on how to use the fireplace, sauna, thermostats and more.

Under the cabin details is a button, which can be pressed to start the job. This enables the user to interact with elements under the button: The tasks can be marked as complete, text entries can be written and another button can be pressed to mark the job as complete.

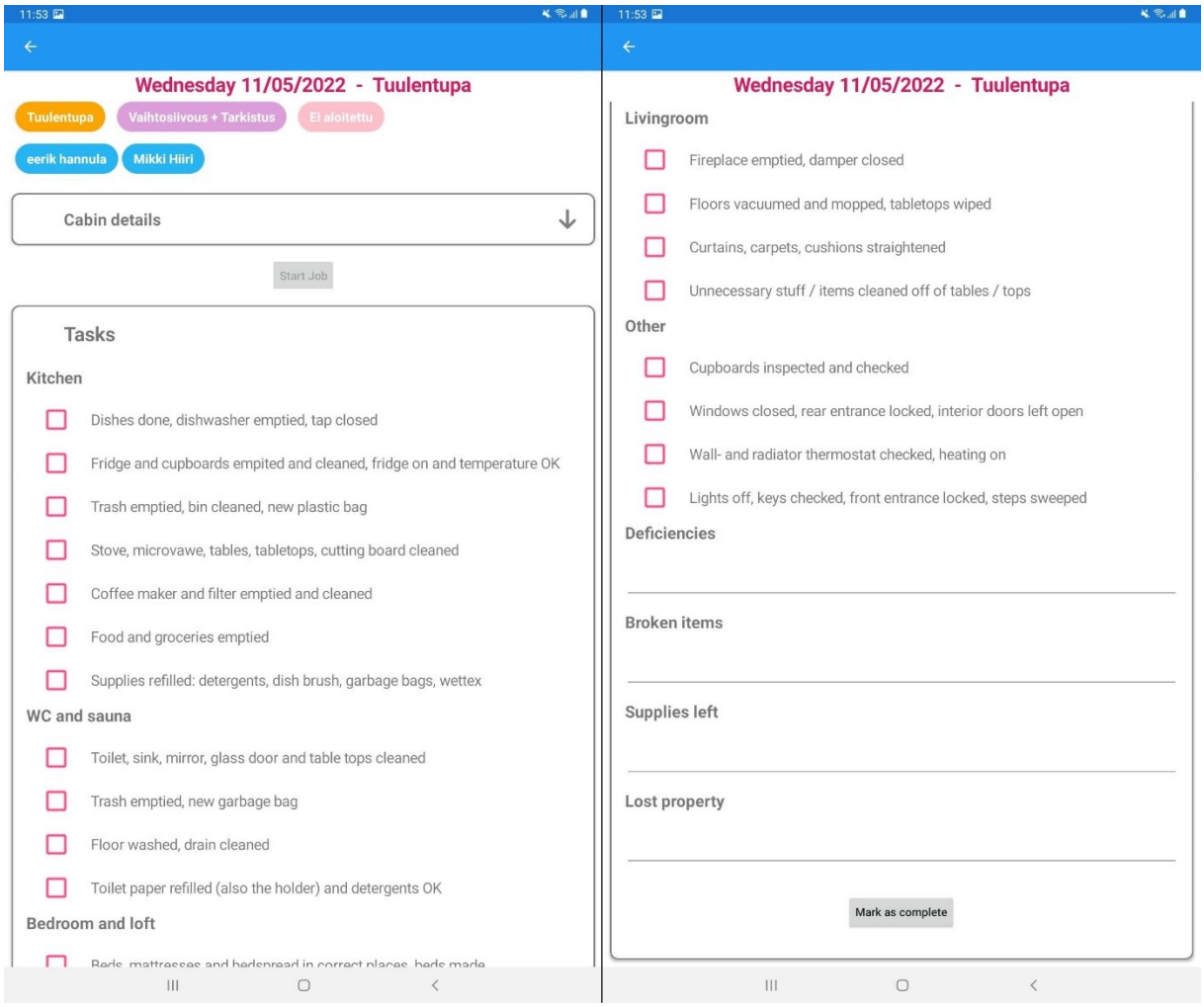

Figure 6. The job page. Upper half on the left, lower half on the right.

4.2 The server environment

Server application was created as an ASP.NET project in Visual Studio. To get started, a tutorial was followed from Microsoft's ASP.NET documentation [12].

4.2.1 Setting up Entity Framework and SQLite database

Before the data can be accessed from the database, or uploaded to the database, the server needs to have a connection to it. This was achieved through Entity Framework. When the application starts, database context is configured to the Entity Framework. This configuration determines the location, type and connection string of the database, which can be seen in Figure 7.

```
0 references
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    var connectionStringBuilder = new SqliteConnectionStringBuilder
    {
        DataSource = "C:/PathToTheDatabase/Database.db"
    };
    var connectionString = connectionStringBuilder.ToString();
    var connection = new SqliteConnection(connectionString);

    optionsBuilder.UseSqlite(connection);
}
```

Figure 7. Configuration method of database context.

Once the server-application has connection to the database, Entity Framework requires specifications about which objects are mapped to the database. This was done by creating a “DbSet<T>”-objects into the database context class. The “<T>”-implies a generic object, which marks the object to be mapped into the database, as seen in Figure 8.

```
6 references
public DbSet<Cabin> Cabins { get; set; }
9 references
public DbSet<User> Users { get; set; }
```

Figure 8. User and Cabin objects that are mapped to database objects through Entity Framework.

4.2.2 Creating controllers

Controller is a class in C#, which controls the way devices communicate with the API. It's responsible for receiving requests and determining which response to send back and what to do when the request arrives.

The controllers were created using Visual Studios built-in tool, which generates some of the essential code automatically. From the Visual Studios solution explorer, developer chooses to add scaffolded item, and then selects "Add API controller with actions, using Entity Framework". This presents a dialog where the developer can determine which object the controller handles, and which database context to use.

After the controller is added, code is generated which contains basic logic for GET, POST, PUT, and DELETE requests. Developer can then adapt this autogenerated code to fit the purpose of the controller.

PUT-request has some glaring issues, when handling large objects with many properties. This is why it was replaced by a more fitting request: PATCH. When updating data using PUT-request, it updates the complete object; all the properties of the object. If two PUT-requests for the same object are received in a close interval, the later request overrides the prior request. This might not be the wanted outcome especially when the requests handle different properties of the object. PATCH-request updates only the property it is meant to update, so no unwanted overrides are performed.

4.2.3 Data transfer objects

When sending data objects between devices, some of the properties might not be wanted or needed on the other device. In these cases, it might be desirable to have separate DTO (Data Transfer Object) to be the container of the data.

DTOs were used in this project to encapsulate transferred data. It was also helpful to have these separate objects from the Entity Frameworks database

objects, because Entity Framework requires that the properties of database objects are structured exactly like the columns in the database. This made the data processing more flexible on the server side.

As seen in the Figure 9, Job-class is used by the Entity Framework to map the object into the database. To save space in the database, only id-properties of the cabin and the employees are saved to the database. When these objects are queried from the database, a converter is used to convert Job-classes to JobDTO-classes, to make them ready for sending inside a response.

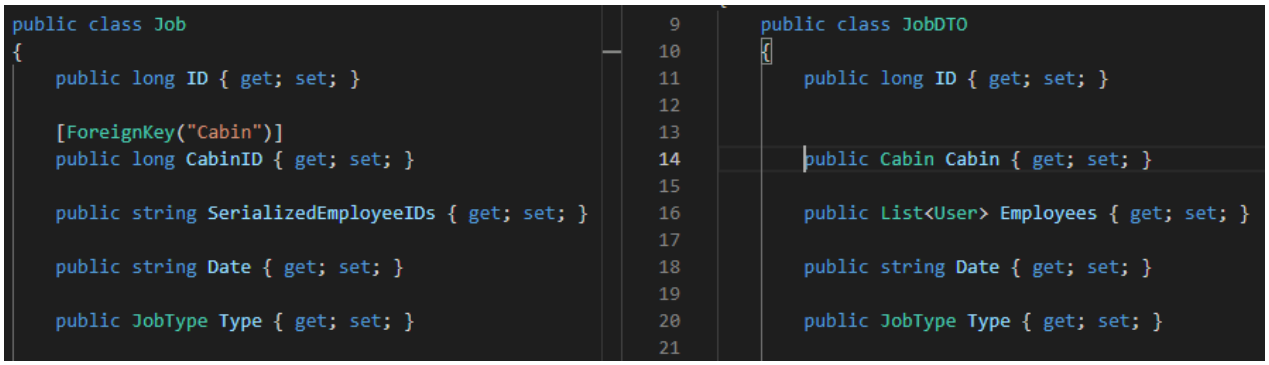

Figure 9. Job (Right) and JobDTO (Left) classes. Job-classes are structured mirroring the database tables, while jobDTO-classes are structured for transfer to the mobile application.

5 Testing

In software development, testing is the process of verifying that the application does what it is supposed to do. In this project, the testing was done concurrently with the development process. The testing was performed against the set of test cases, which are based on the requirements. The debugging happened on a real Android-device; an emulator was not used.

To debug on an android device, it must be setup for development. To achieve this, the build number – visible in device settings – must be tapped 7 times. This makes the developer settings visible, from where the USB-debugging can be turned on.

5.1 Testing the API

Postman [9] is a tool designed to help with API-implementations. With it, the developer can send HTTP-requests for testing purposes to APIs. It's possible to create "Workplaces" in Postman platform, which can then be shared with the development team. Requests can also be saved and grouped, which ensures that requests between projects won't get mixed up.

In this project, Postman was used to test the server's API by sending test requests to it and checking that the returned data matched the expectations. Once the API was working as intended, it was easier to troubleshoot issues because the problems usually resided on the client-side.

5.2 DB Browser for SQLite

DB Browser for SQLite [15] is an open-source tool to work with SQLite database files. It has a clear graphical user interface, which makes it comfortable to use. The user can inspect, add, edit and delete data from the database by writing SQL-queries or using the features provided by the tool.

In this project, the tool was used to create the database structure, and also test if the Entity Framework did what it was supposed to do. It made it easy to see if data was added, edited or deleted. While the project moved forward, changes had to be made to the database structure, which was simple with this tool.

5.3 Efficiency between the client and the server

Measuring the time between request and response between the client and the server was done using .NET class Stopwatch [14]. Stopwatch is used to measure execution time of an application. It can be used globally for the whole application, but in this project, it was used within methods to determine elapsed time between the method start and return.

The largest objects in the application were the cabins. Dummy cabins with large string content were generated to the database and used as a test content when requesting data. The test requests were performed with alternating quantity to see if the queries become slower when enough data is added. The results of this test can be seen in Table 3

Table 3. Time elapsed to get a response from the server, depending on the size of the data.

Quantity of cabin-objects	Time elapsed between request leaving client, and response arriving to the client in milliseconds
0, no data	33.27 ms
10	48.53 ms
100	89.70 ms
1000	399.99 ms

The testing was done in local area network. When deploying the server to the internet, the testing needs to be remade to see how much the results are varied on the public network. From this test can be deduced that the server queries data sufficiently fast even with larger data sets.

5.4 Encountered problems

In software development, complications and problems are unavoidable occurrence. Some of the problems are easily fixed, while others might take a considerable amount of effort to solve. In this subchapter, the most notable unsolved problems are covered.

5.4.1 Android OS theme

The color theme used in the android device is automatically integrated into Xamarin.Forms. This causes the text color to change, depending on whether a light mode, or a dark mode is used on the Android device. The background color of a Xamarin.Forms page is not affected by the device theme, thus making it possible to text color and background color to be indistinguishable.

One solution to this problem would be to manually determine the desirable colors of the elements, but this brings a lot of unnecessary work for the developers. The other solution would be to override the device theme at the application startup, which would be the preferred way.

The override of the device theme was attempted, but no difference could be seen on the UI. Further investigation of the problem is required but left for a later date.

5.4.2 Unstable internet connection

The application heavily relies on a connection to the server. If the connection is unstable or bandwidth of the connection is too slim, the application might

become unusable. To solve this, a system that checks the strength of the connection has to be implemented. This system has to be capable of determining what to do when the connection cannot be established.

A local database on the mobile application might also be needed. This database would temporarily store data, in case there is no internet connection. Once the connection is found, actions would be taken to synchronise the databases on both ends.

6 Conclusions and future work

The goal of the thesis was to develop a mobile application for the needs of Pyhä-Luosto Matkailu Oy. Requirements were set to determine the features wanted in the application. The most essential (Labeled as “Must have”) requirements were fulfilled, making the core system of the project operational.

The user-type specific features were all implemented. The organizer can add, edit and delete cabins, users and jobs. The cleaners are able to inspect their work-shifts, start and end jobs and write reports of the cleaning jobs.

The communication between the client and the server was implemented successfully. It was tested and confirmed as functional, but the testing was done inside a local area network. When the project is published and moved to the public internet, these tests need to be remade to ensure reliable functionality of the application.

Some of the “Should have”-labeled requirements were also implemented. The state of the cleaning can be seen when inspecting created jobs, and the cleaner-accounts can change the state by starting or finishing the job. The application was made multilingual and it currently supports English and Finnish languages.

All in all, the application does what it is meant to do, but the user experience might not be at the expected standard. There are still features to implement and problems to solve, which are discussed in the following and final subchapter.

6.1 Further development

Some of the non-essential requirements did not make their way into the current version of the application. There is no implemented version of a system that calculates the work hours of an employee and returns a clear list of them. The map tool where the cabins can be seen was completely left out of the project,

but it will be added later on. The chat functionality and iOS support were already labeled as “Won’t have” and for now, will remain that way.

The user-interface was made with the requirements in mind. Because of this functionality-centric approach to the development process, the UI was left lacking and not reflecting the finished product. Making the graphical user interface visually pleasing is going to be one of the next steps in improving the user experience of the application.

Even though the user authentication is functional, the credentials used in the authentication process are currently in plain text. This is a serious security flaw, and should be focused on with the highest priority. The passwords should be hashed before they leave the client application. The server stores these hashed passwords, and in login situation the hashed passwords are compared instead of plain text passwords. This reduces the risk of unwanted entities intercepting these credentials in the plain text form.

Finally, it has to be figured out how to publish the project to be used in the real world. The server environment has to be set up so that it can be accessed from the internet with sufficient security measures in place and the release version of the android application needs to be made available to the employees of Pyhä-Luosto Matkailu Oy.

References

1. Pyhä-Luosto Matkailu Oy website [Online]. Available at: https://pyha-luostomatkailu.fi/. [Referred on 21.4.2022].
2. Microsoft .NET Developer platform [Online]. Available at: https://dotnet.microsoft.com/en-us/. [Referred on 21.4.2022].
3. Xamarin – mobile development framework [Online]. Available at: https://dotnet.microsoft.com/en-us/apps/xamarin. [Referred on 21.4.2022].
4. ASP.NET – Web development framework [Online]. Available at: https://dotnet.microsoft.com/en-us/apps/aspnet. [Referred on 21.4.2022].
5. MVVM architectural pattern, Wikipedia [Online]. Available at: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel. [Referred on 21.4.2022].
6. MVC architectural pattern, Wikipedia [Online]. Available at: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. [Referred on 21.4.2022].
7. Visual Studio IDE [Online]. Available at: https://visualstudio.microsoft.com/. [Referred on 21.4.2022].
8. SQLite database engine [Online]. Available at: https://www.sqlite.org/index.html. [Referred on 21.4.2022].
9. Postman API platform [Online]. Available at: https://www.postman.com/. [Referred on 21.4.2022].
10. Entity Framework – object-database mapper [Online]. Available at: https://docs.microsoft.com/en-us/ef/. [Referred on 21.4.2022].
11. MoSCoW- method, Wikipedia [Online]. Available at: https://en.wikipedia.org/wiki/MoSCoW_method. [Referred on 21.4.2022].
12. Creating a web API with ASP.NET – Microsoft [Online]. Available at: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio. [Referred on 22.4.2022].

13. INotifyPropertyChanged - .NET interface [Online]. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.inotifypropertychanged?view=net-6.0. [Referred on 23.4.2022].

14. Stopwatch - .NET class [Online]. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-6.0. [Referred on 13.5.2022].

15. DB Browser for SQLite – a database management tool [Online]. Available at: https://sqlitebrowser.org/. [Referred on 15.5.2022].