Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering

Bachelor Thesis

Developing and Implementing a Health Care System in .NET

Mohammad Zabadi

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Mohammad Zabadi

Informatics

Thesis title

Developing and Implementing a Health Care System in .NET

Objectives of thesis

The thesis aims to analyse the capability and the advantages of developing a health care system unified in a .NET platform, defining methods, functions, frameworks, and platforms within the .NET platform needed for implementing such a system. The project will consist of a mobile application, a database deployed on a web server and a web application.

Methodology

The first section of the theoretical part of the thesis will concern the project description, technologies, and frameworks associated with the project based on the synthesis of the gained knowledge. The second part will touch on the necessity of such a system and further enhancements reinforcing the system functionality.

The practical part will comprise the design and implementation of the project prototype, followed by deploying, testing, and evaluating the system.

The proposed extent of the thesis

35-40 pages

Keywords

.NET, Xamarin, ASP.NET, Blazor, SQL server, QR Code, Health care, Unified

Recommended information sources

Bilgin, Can. Mobile Development With . NET : Build Cross-Platform Mobile Applications with Xamarin. Forms 5 and ASP. NET Core 5, 2nd Edition, Packt Publishing, Limited, 2021.
Himschoot, Peter. Blazor Revealed : Building Web Applications In . NET, Apress L. P., 2019.
".NET: Free. Cross-Platform. Open Source." Microsoft, dotnet.microsoft.com/.
Price, Mark J.. C# 9 and . NET 5 – Modern Cross-Platform Development – Fifth Edition : Build Intelligent Apps, Websites, and Services with Blazor, ASP. NET Core, and Entity Framework Core Using Visual Studio Code, Packt Publishing, Limited, 2020.

Expected date of thesis defence

2021/22 SS – FEM

The Bachelor Thesis Supervisor

Ing. Jiří Brožek, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 1. 11. 2021

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 23. 11. 2021

Ing. Martin Pelikán, Ph.D.

Dean

Prague on 15. 03. 2022

Declaration

I declare that I have worked on my bachelor thesis titled " Developing and Implementing a Health Care System in .NET" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on March 15th ___________________________ Zabadi Mohammad

Acknowledgement

I would like to express my deep and sincere gratitude to my supervisor Ing. Jiří Brožek, Ph.D., for giving me the opportunity to do my work under his supervision and for providing invaluable guidance throughout this work. My completion of this work could not have been accomplished without his advice and encouragement.

Developing and Implementing a Health Care System in .NET

Abstract

This bachelor thesis focuses on using the .NET platform for developing and implementing a healthcare system that allows the patient's medical data to be stored and retrieved from a database. This system consists of a mobile application implemented using the Xamarin platform, a web application implemented using Blazor WebAssembly, and a server implemented in ASP.NET core connected to a database built using SQL server. The thesis also introduces WebAssembly technology and demonstrates the newly introduced .NET framework Blazor WebAssembly.

This work also includes implementing a proposed approach of passing data access permission using a digital key transformed into a QR code.

The frameworks and technologies used for implementing the system are defined in the first part of this work, including an amply description of WebAssembly and Blazor technologies. The second part of the thesis covers the system development phases, including the structure of the implemented system and examples of relevant source codes to illustrate the system implementation phase. The last section of this work discusses some proposed improvements and the benefits of using the presented system.

Keywords: Healthcare, .Net, C#, WebAssembly, Blazor WebAssembly, ASP.Net Core, SQL Server, Xamarin, QR Code.

Vývoj a implementace zdravotnického informačního systému v .NET

Abstrakt

Tato bakalářská práce se zaměřuje na využití platformy .NET pro vývoj zdravotnického informačního systému umožňujícího ukládat a načítat zdravotní dokumentaci pacienta v databázi. Systém se skládá z mobilní aplikace implementované s použitím platformy Xamarin, webové aplikace implementované pomocí Blazor WebAssembly, a serverové části implementované s pomocí ASP.NET Core a využívající databázi SQL Server. Bakalářská práce taktéž představuje technologii WebAssembly a demonstruje technologii Blazor WebAssembly, nově zahrnutou v .NET Frameworku.

Tato práce také zahrnuje implementaci navrhovaného přístupu předávání oprávnění k datovému přístupu pomocí digitálního klíče transformovaného do QR kódu.

V první částí práce jsou definovány frameworky a technologie použité při implementaci systému, včetně popisu technologií WebAssembly a Blazor. Druhá část práce pokrývá všechny fáze vývoje, včetně popisu struktury implementovaného systému a relevantních příkladů zdrojového kódu, které ilustrují implementační fázi. Poslední část práce se pak zabývá návrhem dalších vylepšení a přínosy prezentovaného systému.

Klíčová slova: Healthcare, .Net, C#, WebAssembly, Blazor WebAssembly, ASP.Net Core, SQL Server, Xamarin, QR Code.

Table of contents

List of figures

List of tables

List of abbreviations

AOT – Ahead-of-Time

API – Application Programming Interface

ASP – Active Server Pages

CSS – Cascading Style Sheet

DOM – Document Object Model

EF – Entity Framework

HTML – Hypertext Markup Language

HTTPS – Hypertext Transfer Protocol Secure

IL – Intermediate Language

JIT – Just-in-Time

MVC – Model-View-Controller

MVVM – Model-View-ViewModel

QR – Quick Response

REST – Representational State Transfer

SPA – Single Page Application

SQL – Structured Query Language

UI – User Interface

URL – Uniform Resource Locator

W3C – World Wide Web Consortium

WASM – WebAssembly

XAML – Extensible Application Markup Language

1 Introduction

A patient's medical record is one of the essential elements in the healthcare system; it helps evaluate the patient's health status and accurately determine the diagnosis and treatment. A well-developed system managing the patient's medical records serves the interest of both health institutions and patients.

Based on personal experience, many healthcare institutions in Prague, Czech Republic, are using a poorly developed healthcare system, or in the best case, a system that stores their patients' data only. The idea of developing a shared healthcare system emerged from this experience, allowing several healthcare institutions to share the same system with the same database for retrieving and manipulating the patients' medical records, where the patient data can be accessed only by the patient's permission.

In May 2021, Microsoft released Blazor WebAssembly, giving the possibility to implement a web application in the .NET platform. The chance presented itself to introduce Blazor WebAssembly and implement a health care system in the .NET platform with Blazor WebAssembly as part of it.

This work will describe the mentioned system implementation, alongside a theory description of WebAssembly, Blazor, and other frameworks and technologies used for system implementation.

2 Objectives And Methodology

2.1 Objectives

The thesis analysis the capability and the advantages of developing a healthcare system unified in a .NET platform, defining methods, functions, frameworks, and platforms within the .NET platform needed for implementing such a system. The project will consist of a mobile application, a database deployed on a web server, and a web application. The thesis also introduces WebAssembly and Blazor WebAssembly technologies and describes their part of the implemented system. The system is implemented in the .NET platform. Therefore, Blazor WebAssembly, ASP.NET core, and Xamarin platform are used to implement the three sides of the system. This work proposes using a digital key to pass the data access permission from a patient to medical personnel in an easy and secure approach by displaying the key as a QR code.

2.2 Methodology

The methodology of the thesis is based on studying relevant and reliable scientific sources of information concerning the .NET platform. The primary and most reliable source of knowledge is Microsoft documentation; other sources are books related to the technologies and concepts used.

This work divides into theoretical and practical parts; both parts examine the system's three sides individually. Based on the synthesis of gained knowledge, the technologies and frameworks used are described in the theoretical part of the work. The system's main structure is defined for implementation in the practical part.

The practical part of the thesis concerns the system development phases. The implementation phase illustrates the general structure of the solution and the implementation of the three sides of the system using the frameworks and technologies mentioned in the thesis objectives and several other technologies covered in the theoretical part. The implementation is illustrated by snippet source codes from the work solution.

This work also discusses the benefits and advantages of the proposed system implementation, including proposed improvements that can be applied to the system.

3 Literature Overview

3.1 Client-side

Before diving into the concepts of both Blazor and WebAssembly, a brief recall on web development from a pre-Blazor time will help illustrate these two technologies.

JavaScript might be the first word that comes to a developer's mind when speaking of web development. The reason is that 97.8% of all the websites are using JavaScript as a client-side programming language, according to the latest w3techs report (1). Many libraries and frameworks using JavaScript were developed throughout the years, allowing us to build faster and richer web apps, such as Angular, React, and Vue.js. Such frameworks can also use a higher-level like TypeScript, which later is transpiled into JavaScript that will run the actual code on the browser. (2)

A big side of the competition between modern browsers was focused on JavaScript performance, using various JavaScript engines, with different optimization features like just-in-time (JIT) compilation. For JavaScript code to be executed, JS file (or inline) needs to be downloaded into the browser, parsed, then passed through a compiler (since JavaScript is an interpreted language), which translates the source code into a bytecode for the machine to understand and execute it, with JIT compilation, JavaScript is compiled to executable bytecode at run time. (3)

Since March 2017, when WebAssembly was first released, other programming languages could run within the browser, which led to JavaScript not being the only language to do so.

3.1.1 WebAssembly

Webassembly.org, the official site, defines WebAssembly as: *"WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. WASM is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications."* (4)

Simply put, WebAssembly is a new technology providing a way for a code written in other programming languages than JavaScript, such as C++ and C#, to run in the modern browsers near-native speed.

As mentioned before, for JavaScript code to be executed, the code file needs to be downloaded into the browser. The browser then takes care of parsing, optimizing, and JIT compiling of the code, but when it comes to WebAssembly, parsing and compiling is done on the server, where the code is compiled in a format called WASM, the compiled file is then downloaded into the browser, where it gets just-in-time compiled. (2) The difference between JavaScript and WebAssembly execution processes is illustrated in *Figure 1*.

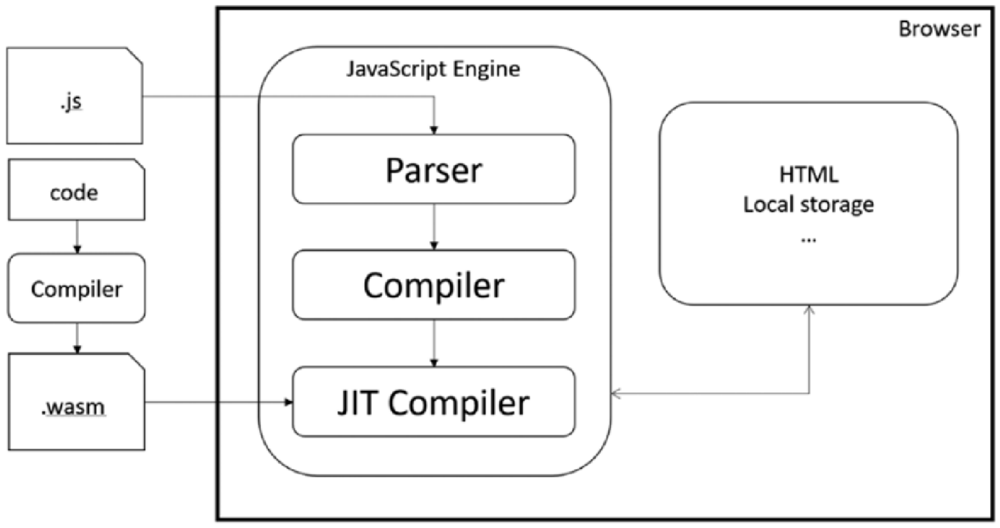


Figure 1: JavaScript versus WebAssembly execution process (2)

W3C Community Groupe develops the open standards for WebAssembly; it was first introduced in 2015, the first official release of WebAssembly was in March 2017 by Firefox, making it the first major browser to support WebAssembly, followed by chrome and opera in the same month, and today, 94.7% of all browsers support WebAssembly (5). Nowadays, WebAssembly is co-developed in coordination between W3C, and major IT companies, like Microsoft, Google, Apple, and many others. In addition, many individual developers from all around the globe are contributing to WebAssembly every day. (4)

It is crucial to understand that WebAssembly is not intended to replace JavaScript. Instead, WebAssembly is designed to complement JavaScript and work in conjunction with it. Combining JavaScript and WebAssembly's strong points allows developers to build better performance and faster web apps. (6)

WebAssembly technology is currently being designed as an open standard inside the W3C WebAssembly Community Group, including engineers from all major browsers. During its design, W3C focuses on achieving four main goals: (4)

- Efficient and fast: WebAssembly is designed to allow code written in any language to run within the browser and be executed at near-native speed by taking advantage of common hardware capabilities common to all contemporary hardware; it decodes, validate, and compile the code file so efficiently, equally to JIT or AOT compilation.
- Safe: WebAssembly code executes within a memory-safe, sandboxed execution environment. WebAssembly models enforce the web browser same-origin, permissions, and security policies when embedded in the web. Additionally, WebAssembly allows no direct interaction between the module and browser's DOM; the only way to do so is by using JavaScript interop.
- Open and debuggable: even though it is a low-level assembly language, WebAssembly text format is pretty printed in a human-readable way, allowing developers to write, view, optimize, and debug the code by hand.
- Part of the open platform: WebAssembly is designed in a way not to break the web but to interoperate with other web technologies and maintain the versionless, feature-tested, and backwords-compatible nature of the web. WebAssembly modules can communicate with JavaScript context, same as JavaScript; these modules, through the web APIs, can access the browser functionality.

3.1.2 Blazor

"Blazor is a framework for building interactive client-side Web UI with .Net." (7). The word Blazor is a combination of words Browser and Razor. With Blazor, we can create rich interactive UIs in C# instead of being forced to use JavaScript, in other words, develop a web app front-end logic using C#, HTML, and CSS using Razor syntax that supports binding, events, dependency injection, and many more techniques for building robust web apps. Also, Blazor allows us to share the server-side and client-side app logic written in .NET. (7)

3.1.2.1 Razor components

Blazor apps are component-driven apps; they are built using components known as Razor components, informally referred to as Blazor components. Components are .NET C# classes implemented using Razor syntax, which is a combination of C#, HTML, and Razor markup in the same file. *"A component is a self-contained portion of user interface (UI) with* *processing logic to enable dynamic behavior. Components can be nested, reused, shared among projects, and used in MVC and Razor Pages apps."* (8) Razor components are the base element of UI in Blazor apps, defining flexible UI rendering logic and handling user events. (8)

Razor components can be nested, meaning that a child component can be used in a parent component using HTML syntax, where the component name is used as an HTML tag, allowing a component to be reused within Blazor application or even be shared and distributed as Razor class libraries or NuGet packages. (7)

Razor components are defined using Razor syntax, dividing the component into markup and code blocks. The markup block determines how the component and component elements are parsed or functions, providing a way to bind data and events, nest child components, define routing, and many more. The code block allows specifying components state by handling events, data, and customizing components logic. (8)

3.1.2.2 Blazor hosting models

Currently, Blazor has two hosting models; the first one is Blazor Server, which was the first to release by Microsoft in September 2019, the second is Blazor WebAssembly, released in May 2020.

Blazor Server

As the name implies, in the Blazor Server hosting model, the web app (Razor components specifically) is executed on the server built within an ASP.NET Core. The server communicates with the browser over SignalR Connection, sending UI updates, event handling, and JavaScript calls between the server and the browser. (9)

ASP.NET Core SignalR is an open-source library adding real-time web functionality to apps (10). In our case, it enables Blazor Server to receive UI events from the browser and send back UI updates from the server to the browser instantly. Real-time communication is handled using WebSockets, Server-Sent Events, or Long Polling, and depending on the capabilities of both server and clients, SignalR automatically chooses the best transport technique. SignalR connection is established on the client-side by Blazor Script (blazor.server.js). (11) Blazor Server hosting model is illustrated in *Figure 2*.

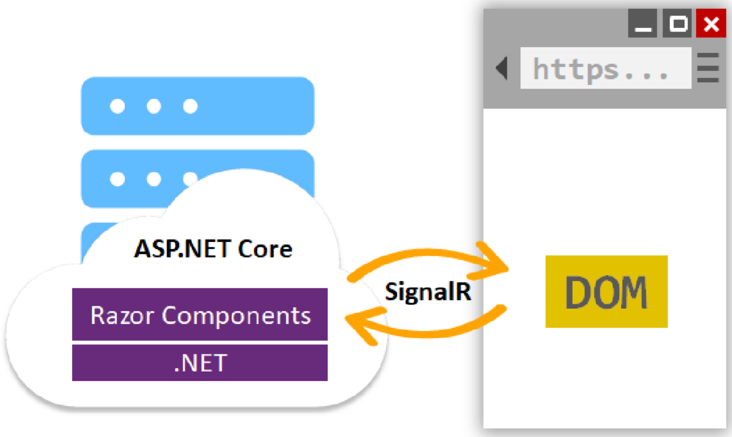


Figure 2: Blazor Server (7)

In traditional ASP.NET core apps that use Razor views or Razor pages, the entire page is rendered on client request, meaning that every line of Razor code in Razor page or view emits HTML in text form. When the client sends a request to the server, the server renders the entire page to HTML again and sends it back to the client. However, Blazor server apps render the page more efficient. Since Blazor apps are based on Razor components, a Blazor server app renders the page by components, not the entire page. (7)

The components are rendered into a so-called render tree, which is a binary representation of Document Object Model (DOM), which includes sate held in properties and fields. After the components are rendered on the client, the components can be updated by app events and user interaction, and when the update occurs, the render tree calculates the UI difference, the difference is sent to the client over SignalR in a binary format to update the DOM, where the browser updates the page accordingly. (3)

Benefits of Blazor Server hosting model: (9) (11)

- The initial load time for a Blazor Server app can be much less than a Blazor WebAssembly app due to the smaller download size of Blazor Server apps.
- Blazor Server apps can run in browsers that don't support WebAssembly.
- Since the app is running on the server, the app can take full advantage of server capabilities.
- The Blazor server app's code stays on the server and is not served to the client, preventing decompilation of the code.

Limitation of Blazor Server hosting model: (9) (11)

- The app must be hosted on an ASP.NET core server.
- Blazor Server apps do not support offline mode.
- Usually, Blazor Server apps have higher latency since every UI update requires a network roundtrip to the server.
- Since Blazor Server apps rely on SignalR for every UI update, the scaling might be challenging and requires more server resources.

Blazor WebAssembly

In Blazor WebAssembly hosting model, the Blazor app runs in the browser on a WebAssembly-based, meaning that the browser must support WebAssembly, and since Blazor WebAssembly uses open web standards, no plugins are required for it to run in the browser. (7) Blazor WebAssembly hosting model is illustrated in *Figure 3*.

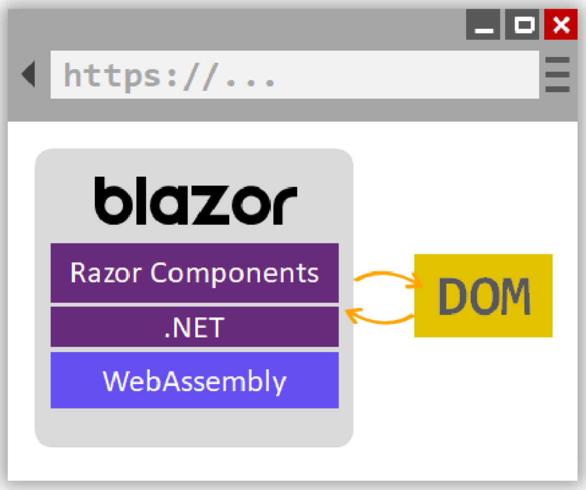


Figure 3 Blazor WebAssembly hosting model (7)

Blazor WebAssembly is a single-page application (SPA) framework. A single-page application is a web application design that allows dynamically rewriting and updating the body content of a web page in response to user interactions, eliminating the need to load the entire new page, which results in enhanced app performance and faster transitions, and allowing the app to run at native speed. (9)

When a Blazor WebAssembly application is built and run in a browser, the Blazor app, its dependencies, and the .Net runtime are downloaded to the browser (this is handled by blazor.weassembly.js script), the app is then executed on the browser UI thread, and the

UI updates and app events are handled within the same process. Blazor WebAssembly does not access DOM directly and can only update the DOM via JavaScript interop. (11)

Blazor WebAssembly does not necessarily require a server and can be deployed without it. If the app is deployed without an ASP.NET core server, it is called a standalone Blazor WebAssembly app. However, some web apps may require one for data access and authentication. If the Blazor WebAssembly app is deployed with an ASP.NET core server, it is called a hosted Blazor WebAssembly app. *"Using hosted Blazor WebAssembly, you get a full-stack web development experience with .NET, including the ability to share code between the client and server apps, support for prerendering, and integration with MVC and Razor Pages."* (11)

The big concern here is the size of the published Blazor WebAssembly app, which is significantly bigger when compared to a Blazor Server app. Keep in mind here that Blazor WebAssembly is a single-page app, meaning that the whole site is downloaded to the client. This negatively affects Blazor WebAssembly performance since it takes longer to download it into the browser. To reduce the download size and speed up the load time, Blazor WebAssembly uses the following strategies: (7) (11)

- Intermediate Language (IL) trimming is performed on the published app to strip out the unused code, reducing the published output's size.
- HTTP responses are compressed.
- Cashing .NET runtime and assemblies (the compiled C# and Razor files) in the browser.
- Support ahead-of-time (AOT) compilation, where the .NET code is compiled directly into WebAssembly.

Benefits of Blazor Server hosting model: (9) (11)

- Blazor WebAssembly apps rely on the client resources and capabilities, resulting in less load on the server.
- Supports offline mode. If the server goes offline, the app remains functional.
- Hosting a Blazor WebAssembly app without an ASP.NET core web server is possible.

Limitation of Blazor Server hosting model: (9) (11)

- The large download size of Blazor WebAssembly apps means a longer time for the initial load of the app.
- Browser capabilities can limit the Blazor WebAssembly app's performance.
- A browser supporting WebAssembly is required to run Blazor WebAssembly app.

Table 1: Comparison between Blazor Server and Blazor WebAssembly (7)

Feature	Blazor Server	Blazor WebAssembly
Complete .NET API compatibility	Yes	No
Direct Access to sever source	Yes	No
Small payload size with fast initial load time	Yes	No
App code secure and private on the server	Yes	No
Static site hosting	No	Yes
Run apps offline once downloaded	No	Yes
Offloads processing to clients	No	Yes

3.2 Server-side

3.2.1 ASP.NET Core

ASP.NET Core is an open-source framework developed by Microsoft for building modern, cloud-enable, and internet-connected applications designed for high performance and that run across platforms on Windows, Mac, and Linux. Using ASP.NET Core, we can build web applications and services, internet of things (IoT) apps, and mobile backends. ASP.NET Core apps run on .NET Core runtime. (12)

It is essential to differentiate ASP.NET Core from ASP.NET. ASP.NET Core is a redesign of ASP.NET; it includes better support of modular architecture. While ASP.NET runs only on the .NET framework, therefore, runs on Windows only, ASP.NET Core can run on both .Net Core and .NET framework, and it runs cross-platform. (13) This is illustrated in *Figure 4*.

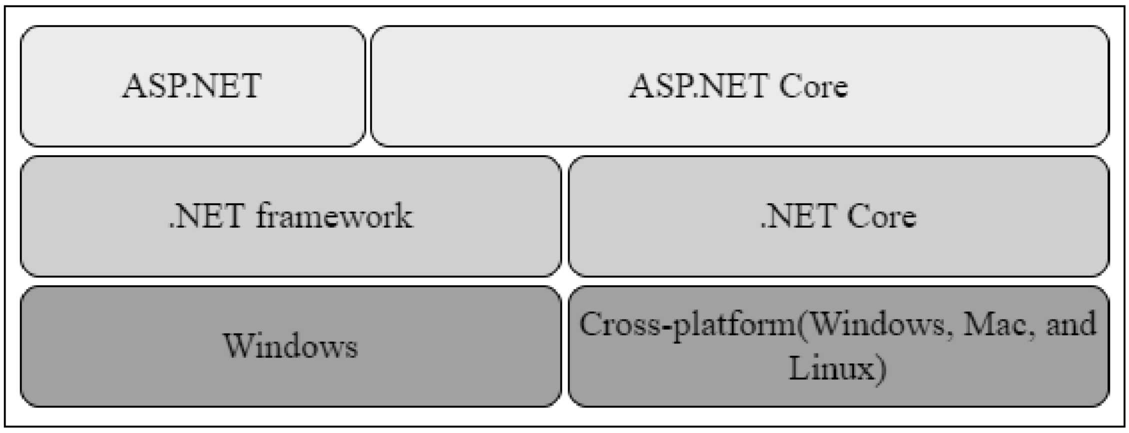


Figure 4: ASP.NET vs. ASP.NET Core (13)

3.2.2 API

Application Programming Interface is a set of protocols and rules that allows applications to communicate with each other for data exchange purposes.

Web APIs are remote APIs for applications to communicate over the internet network using HTTP protocols. In other words, a mobile application can communicate with a web application using web APIs implemented in a web server.

A REST API or RESTful API is the architectural implementation of the API in an application (mobile app or a Web app) that allows the applications to send HTTP requests over the internet for data exchange. (14)

Figure 5 illustrates the design of a web API implementation with ASP.NET Core. MVC and its pattern will be demonstrated in the following section.

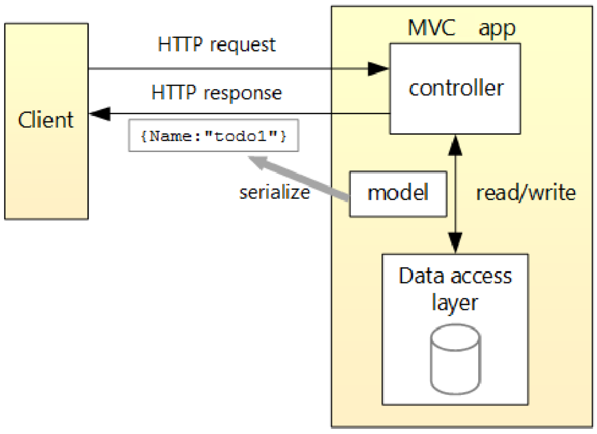


Figure 5: ASP.NET core web API design (15)

3.2.3 MVC

ASP.NET Core uses MVC architectural pattern to separate the server-side application into Models, Views, and Controllers, thus allowing a better division of the code and application functions to help with testing, maintaining, and evolving the application. (16)

The Controller is responsible for receiving requests and sending responses by working with the model component to perform actions and retrieve the data. The Models manage the data structure and business logic, while the View defines how the data received from the Controller is displayed to the user. (16)

In the implemented system, the views will be handled by the client-side. Therefore, this work will not include this component in the MVC pattern.

3.2.4 Repository pattern

One of the popular patterns used in ASP.NET MVC. The repository pattern creates an abstraction layer that mediates between the data source and the business layers. Repositories are C# classes that encapsulate the logic that retrieves the data from the data source and separate the business logic from direct interaction with the underlying data source. This pattern is implemented in the practical part providing a better illustration of its structure and functions. (17)

3.2.5 Entity Framework Core

Entity Framework Core (EF Core) is a modern object-relational database mapper for .NET. EF Core can work with several databases, like SQL Server, MySQL, PostgreSQL, and many more. EF Core and EF are not the same; EF Core is a lightweight, extensible, open-source, and cross-platform version of EF data access technology. Some features are implemented in EF Core but not in EF, and vice versa. However, Microsoft recommends using EF Core on .NET Core for developing new applications. EF Core, in most scenarios, has better performance than EF, including the support for data access code evolvement and implementing new features. (18) (19)

EF Core allows us to work with a database using .NET objects, eliminating the need to write most of the data-access code manually. For data access, EF Core uses models consisting of entity classes and a context object that allows querying and saving data. EF Core could generate a model from an existing database or create a database from a model using EF Migration. When a model changes, Migrations can evolve the database accordingly. (20)

3.2.6 SQL Server

SQL Server is developed, maintained, and marketed by Microsoft; it is a relational database management system (RDBMS). It has the primary function of storing and giving access to the data stored when requested by other software applications on the same computer or over a network. It is built based on SQL and is tied to Transact-SQL (T-SQL). SQL Server has the advantage over other databases in high performance and enhanced security. (21)

3.3 Mobile application

3.3.1 Xamarin

Xamarin is an open-source platform developed by Microsoft for building cross-platform mobile and Windows applications with .NET. Xamarin apps are cross-platform because Xamarin allows us to share about 90% of the application business logic written in C# across platforms. Since Xamarin is built on top of .NET, it allows us to build the application in a managed environment that handles garbage collection and memory allocation with underlying platforms. (22)

It is essential to mention Xamarin.Essentials, which is a library that provides developers with single cross-platform APIs and allows us to access native functionalities such as permissions, device info, file system, and many more. (23)

Xamarin.Forms allow us to create application UI. It is an open-source UI framework for building applications with a single shared codebase across the three major platforms of iOS, Android, and Windows. The user interface is created in XAML with C# in the backend. UI elements are rendered into native controls on each platform. Xamarin.Forms provides many features for building robust applications, such as XAML user-interface language, Databinding, effects, and styling. (24)

XAML stands for eXtensible Application Markup Language; it is based on XML. Therefore, the structure of XAML files comprises elements, attributes, and namespaces. XAML is used for creating an application UI, mainly in Xamarin apps and WPF. (25)

3.3.2 MVVM pattern

MVVM (Model-View-ViewModel) is an architectural pattern that separates an application business and presentation logic from its UI. Simply put, the MVVM pattern extracts the code-behind of an implemented user interface in the view component to a different component called View Model, thus making the application easier to test, maintain, and evolve. The general structure of the software consists of View, View Models, and Models components. (26) *Figure 6* illustrate the relationships between MVVM components.

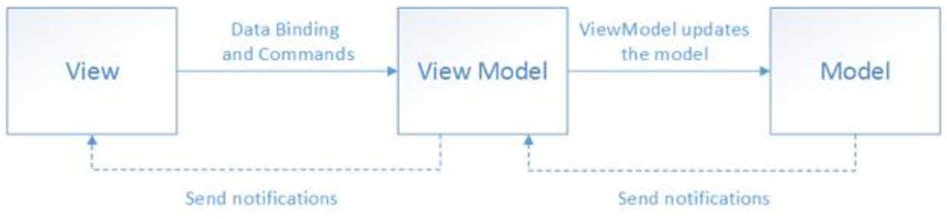


Figure 6: Relationships between MVVM components (26)

The implementation of the pattern is illustrated later in the practical part, giving a better understanding of how the components communicate with each other.

3.4 Shared concepts

3.4.1 .NET

.NET is an open-source development platform for developing and building cross-platform apps using various platforms and frameworks within .NET. .NET allows sharing codes, logic, and functionalities among different apps, making it easier to build an entire system unified in .NET. (27)

The presented work is built in .NET 5, which is an implementation of .NET and the last released version of .NET Core. The word “Core” is not part of the name to emphasize that this is the leading implementation of .NET going forward. .NET Core is the successor of the .NET Framework. However, it does not replace it. The key difference between .NET and .NET framework is the supported platforms. As illustrated in figure 4, the .NET framework supports only windows apps, while .NET supports cross-platform apps. (28)

3.4.2 C#

C# is one of three programming languages supported by .NET and a C family of languages child. It is a modern, object-oriented, and type-safe programming language.

C# supports the four basic principles of object-oriented programming: abstraction, encapsulation, inheritance, and polymorphism. (29) C# also includes the support for component-oriented programming. Modern applications such as Blazor applications are built based on components. Components can be described as packages of functionalities presented as models with properties, methods, and events. C# provides language constructs to support using and building software components. C# supports type safety; it does not allow uninitialized variables in the code to perform unchecked type casts or arrays indexed beyond their identified bounds. (30)

C# offers several features that aid in constructing powerful and robust applications, such as nullable types, lambda expression, LINQ, asynchronous operations, and many more. Programs and libraries written in C# can be maintained and evolve in a compatible manner due to the versioning that C# emphasizes. (31)

3.4.3 HTTPS

Hypertext Transfer Protocol (HTTP) is the foundation of the modern global internet, providing software with a common language to talk and communicate between each other and exchange data in the form of requests and responses. The “S” in HTTPS stands for “Secure.” HTTPS is an extension of HTTP that uses SSL protocol to encrypt HTTP requests and responses, making it more secure than traditional HTTP. (32)

The client requests a web resource hosted on a specified web server using an HTTP request message containing a request command called method, and the server performs the actions needed based on the method received and sends an HTTP response message to the client with a status code indicating the status of the client request. (32)

The client using HTTP methods can retrieve or manipulate the data in the web resource. The most used methods are: GET (retrieve data), PUT (update data), POST (retrieve sensitive data or add data), and DELETE (delete data). HTTP status code is a three-digit numeric code informing the client if the request was successful or not, and in most cases, the reason for failure. Some well-known status codes are: '200' for a successful request, '404' for not found, and '401' unauthorized. (32)

URL (uniform resource locator) is used to specify the location of a web resource on a specified server. URL is used by the client to address the HTTP request; it consists of a scheme, address, and resource name. The schema describes the protocol used to access the resource (like HTTP), the address part contains the server internet address, and the resource name specifies the resource requested on the server. (32)

3.4.4 ZXing

"ZXing ("zebra crossing") is an open-source, multi-format 1D/2D barcode image processing library implemented in Java, with ports to other languages." (33)

ZXing.Net is a port of the ZXing library. ZXing.Net is a library allowing .Net developers to decode and generate various barcodes, including QR Codes, it is supported on many .Net platforms, and the Xamarin platform is one of them.

4 Practical part

This chapter will describe the process of planning, analyzing, designing, implementing, and finally testing the healthcare system software. This approach was inspired by system development life cycle (SDLC) management. (34)

4.1 Motivation and planning

The idea of developing a healthcare system was based on personal experiences. A couple of years ago, I had a health condition, where I had to visit several health institutions, and I had this pile of medical reports that I carried when going to a different institution. Also, in every institution, I was asked to provide the same data I already provided in the previous institution I had been to. I also had a similar experience when I needed to fix a tooth.

A couple of years later, Blazor WebAssembly was released, and since I had a background in C# (all thanks to Czech University of Life Sciences Prague (ČZU) and particularly Mr. Ing. Jiří Brožek, Ph.D.), I started thinking about developing and implementing a prototype of this system in .NET. After researching, I found out that it is possible to implement such a system with:

- Xamarin for the mobile application. The mobile application allows patients to grant medical personnel access to their data.
- ASP.NET Core for the server-side for data and access management.
- Blazor WebAssembly for the client-side for medical personnel to access, display, and modify the patient data.

All I needed was some effort to gain the knowledge required to implement the system's prototype.

4.2 Analysis

In this stage, a general analysis of the project is performed, defining the requirements, user goals, and limitations for implementing the system. Requirements will be divided into functional and non-functional. Functional requirements specify what the system should do, while non-functional specify how the system performs a specific function. The functional requirements are then split into mobile application, server-side, and client-side. The general shape of the system will be defined according to these requirements.

4.2.1 Functional requirements

- Authentication

  This prototype will not cover the registration of users into the system, and only sign-in operations will be implemented to bring the system closer to reality, where patients and medical personnel are registered into the system by the system admins (health ministry or health institution administration for instance). This work will include the implementation of authentication functionality only on the mobile application. Patients will be authenticated by the mobile application to access their profile on the server.

- Data access

  For data access, the author decided to choose a digital key approach. This key is to be generated on patient request and given to medical personnel to grant them access to patient data. The requirements for this approach are:

  - Requesting a key. The patient using the mobile application will request a one-time-key from the server.
  - Generating a key. The server will generate the key on request and send it back to the patient.
  - Using the key. The medical personnel using the client-side web app will receive the key from the patient and use it to access the patient's data.

- Data retrieval

  After the patient is authenticated to the mobile application, selected personal data will be displayed on the profile view. On the client-side, when the patient gives the medical personnel access, the system will retrieve and display the patient medical records and data.

- Data manipulation

  On the client-side of the system, the medical personnel will be able to:

  - Add new medical records to the patient medical history.
  - Modify patient medical data known as "Anamnesis." This part is different from the medical record.

4.2.2 Non-Functional requirements

- Database

  SQL Server will be the database used to save the system data.

- QR code key

  As mentioned before, the key will be in a digital form. Therefore, for the client-side to receive the key easily and efficiently, the key will be converted by the mobile application to a QR code, where it will be scanned on the client-side.

- Publishing

  Both server-side and client-side will be published on the Azure portal.

4.3 Design

This phase defines the system's architecture, which is built based on the requirements provided in the analysis phase. This includes the design of the logo, communication model, data model, and wireframes for both mobile and client-side.

4.3.1 Name and logo

The auth named the system “QRCARE,” the name is a combination of QR code and healthcare. The same combination is used for designing the system logo; the logo is a combination of a QR code and a red cross that stands for healthcare.

The two logos designed are visible in *Figure 7*.

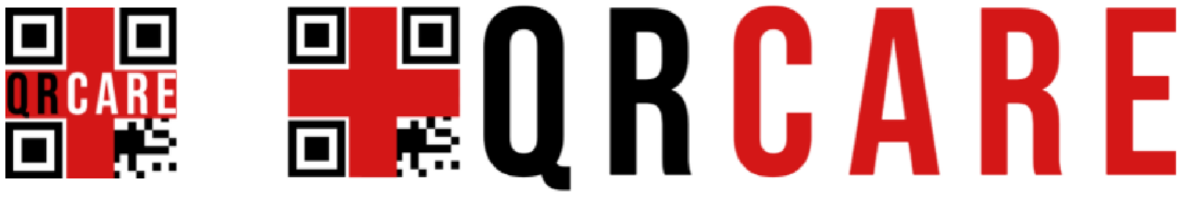


Figure 7: QRCare logo (By author)

4.3.2 Communication model

Data access is granted to medical personnel by patients using a one-time, on-request digital key presented as a QR code; This process is managed through a server. *Figure 8* illustrates the communication model for the data access process, followed by an explanation of the process steps. For simplicity, this communication model will only cover data access management.

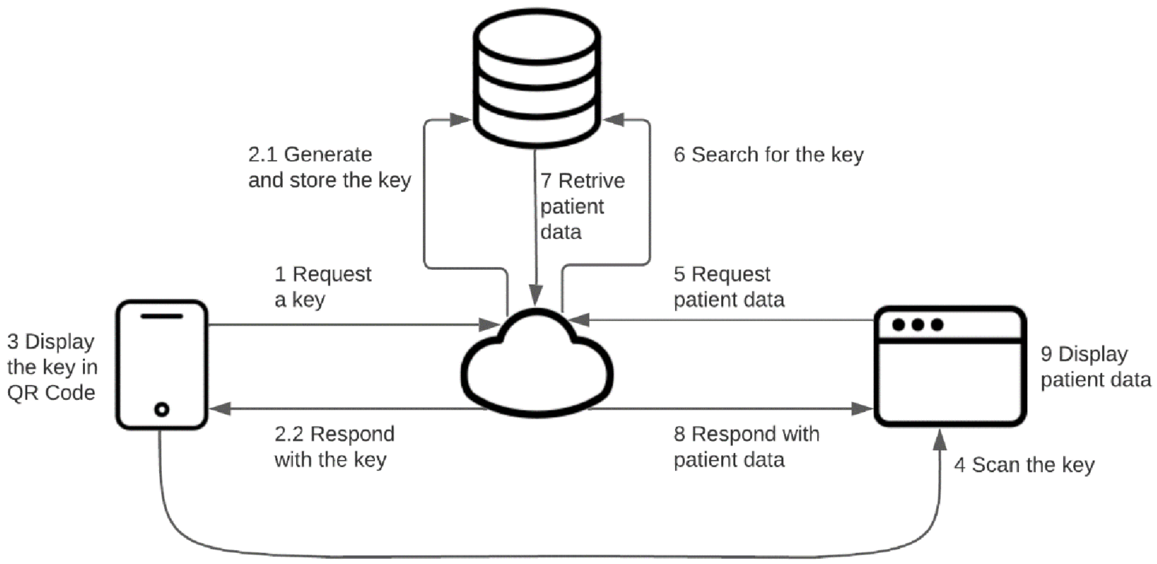


Figure 8: Communication diagram for accessing a patient's data (By author)

1. A Patient sends a request for the key to the server via a mobile app.
2. The server receives the request, generates the key, stores it in the database, and then sends the key back to the patient as a response.
3. The mobile application receives the digital key and displays it in the view as a QR code.
4. A patient presents the code to medical personnel; the code is then scanned on the web app.
5. Web app sends the decoded key to the server and requests to access the patient's data.
6. The server searches through the database for the key received.
7. If the database holds the key, the server retrieves the patient's data holding the key.
8. The server sends back the patient's data to the client.
9. The client displays the data on the browser for medical personnel to examine and manipulate the data.

4.3.3 Data model

A medical background is required to build a healthcare data model close to the real world. The author consulted a health professional and was advised to build the data model based on information provided by the trusted Czech *WikiSkripta* (35).

This data model describes the entities and relationships between them, alongside the attributes for each entity. The model represents the structure of the database of the system. Based on it, the database will be implemented later in the implementation phase based on this model. *Figure 9* illustrates the data model designed.

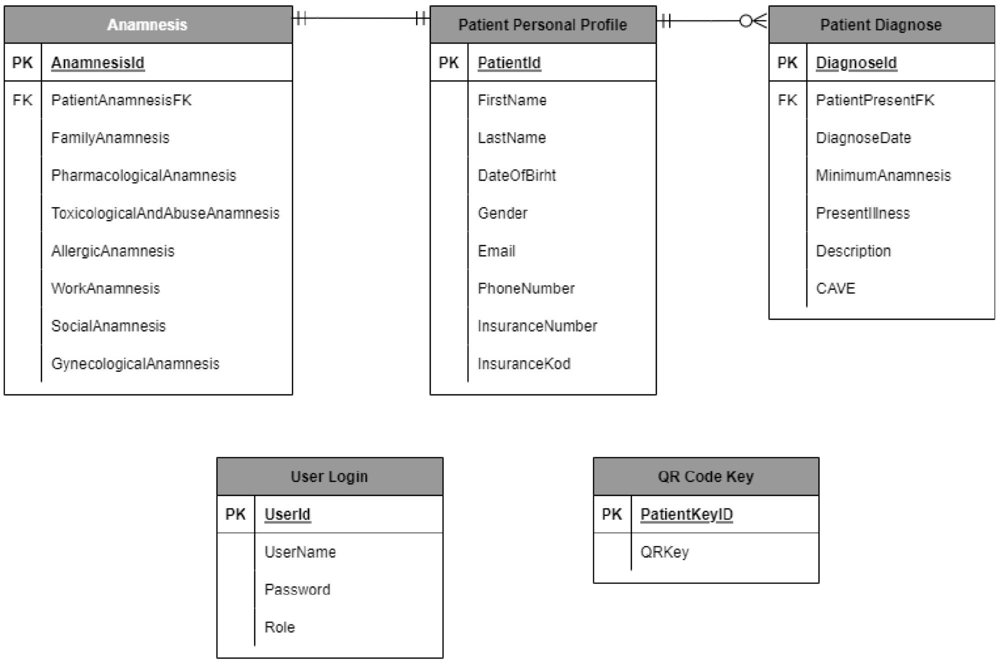


Figure 9: Database model (By author)

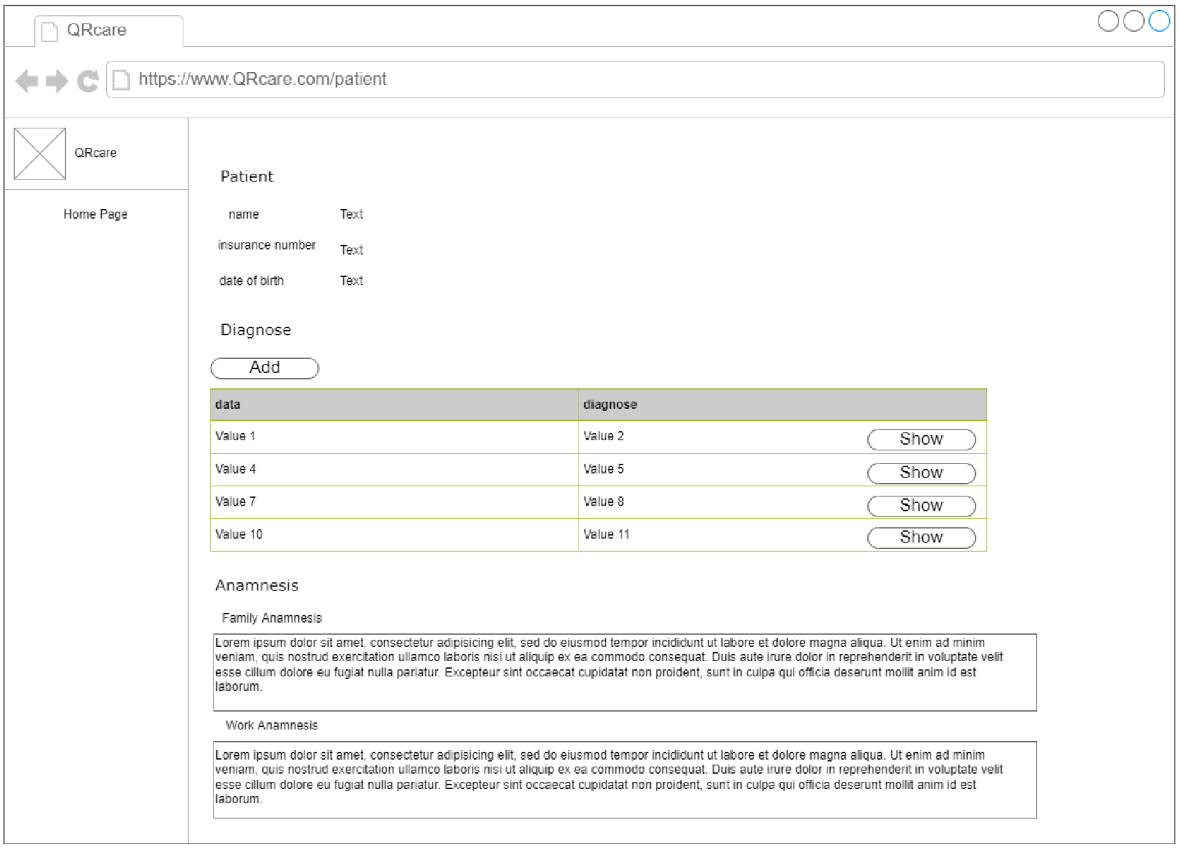
It is essential to point out the following:

- This model covers only a tiny part of the real-world healthcare system. Many entities and attributes needed are missing, like treatments and dental records.
- Data normalization is not fully applied for the introduced data model. In particular, the attribute “InsuranceKod” in the “Patient Personal Profile” table should be separated into a different table for a more accurate model. The same applies to “Role” in “User Login.”
- “User Login” and “QR Code Key” entities have no direct relationship with the “Patient Personal Profile” entity. This approach is not based on any scientific resource; it was elicited from personal reasoning. This approach aims to improve the system’s safety by isolating the sensitive data and separating it from the retrieved data based on a client request. The primary key in all three entities is the same, meaning that a patient will have the same Id in all of them, which defines a 1:1 relationship implicitly. The identical primary key in all three tables can be ensured

by implementing the functionality of creating a new record in the "PatientPersonalProfile" table in a way that new records with the same Id are inserted automatically in the "UserLogin" and "QRCodeKey" tables.

4.3.4 Wireframe

Wireframe creation took part in the design phase. The web application and mobile application layout are visualized using wireframes; this includes the interaction elements placement in the pages and views. This work will only illustrate the patient page in the web application and the "Home view" and "QR Code view" for the mobile application. The patient page of the web app consists of 3 main components: patient personal information, diagnoses, and anamnesis. The button "Show" in the table in the diagnosis's component will open a modal dialog containing additional information for the selected diagnosis. The other page in the web application is the "QR key reader page." As for the mobile application, the patient can ask for a key using a password or via fingerprint authentication on the main page. Other views include the login view and the profile view. The wireframe is illustrated in *Figures 10 and 11*.



Patient

name	Text
insurance number	Text
date of birth	Text

Diagnose

data	diagnose	
Value 1	Value 2	Show
Value 4	Value 5	Show
Value 7	Value 8	Show
Value 10	Value 11	Show

Anamnesis

Family Anamnesis

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Work Anamnesis

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 10: Patient page wireframe (By author)

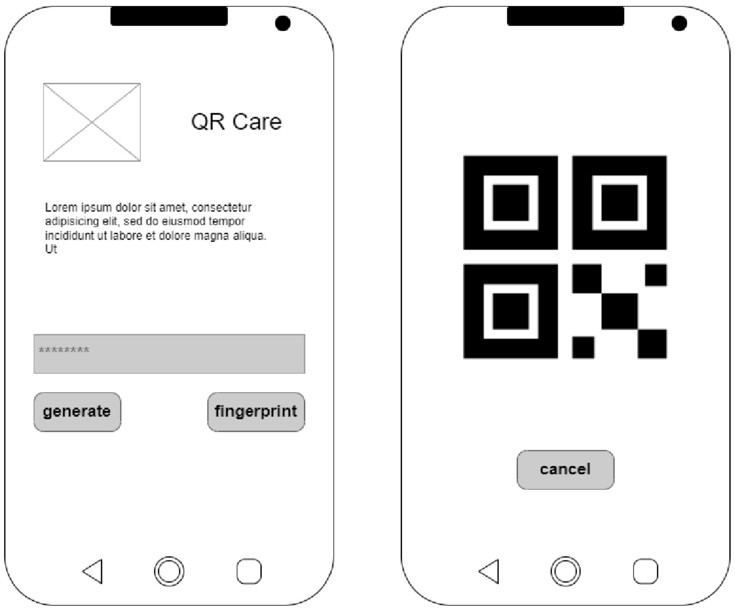

QR Care
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
****
generate
fingerprint
cancel

Figure 11: Mobile application wireframes (By author)

4.4 Implementation

The implementation of the healthcare system prototype will be split into three parts. The first part focuses on developing the server-side of the system handling web API and database connection. The second part covers the mobile application, and the last part describes the implementation of the client-side of the system.

The solution will be split into six main projects, as shown in *Figure 12*. Note that 3 of these projects belong to the mobile application part of the system, while `QRcare.API` project is the server-side, `QRcare.Models` project is the data models and `QRcare.WebApp` project is the client-side of the system.

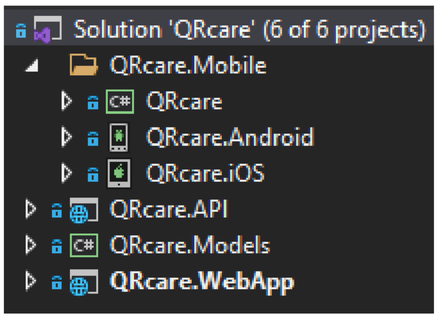

Solution 'QRcare' (6 of 6 projects)
- QRcare.Mobile
  - QRcare
  - QRcare.Android
  - QRcare.iOS
- QRcare.API
- QRcare.Models
- QRcare.WebApp

Figure 12: Solution structure (By author)

Microsoft Visual Studio Community Edition 2019 development environment is used for developing all three parts of the system.

4.4.1 Server-side

ASP.NET Core MVC framework is used for building the RESTful API server-side of the system. The pattern used for implementing the server-side is Repository Patterns, so the server-side consists of three layers, data access layer, repositories layer, and controller layer. The three layers will be described in the following sections.

4.4.1.1 Data access layer

The data model was designed to define entities, attributes, and relationships in the analysis phase. The designed data model will be implemented in this phase using the *"Code First"* approach to Entity Framework. With this approach, we write C# classes that correspond to data model entities, and EF will handle the creation of the database. C# classes represent the model entities, classes properties represent the attributes, and navigation properties define the relationships between entities.

Source code 1 is the `PatientPersonalProfile` class source code that will serve as an example of how the classes are built. The class name will be assigned later as the table name in the database by EF. This class contains several properties, and each property will be translated to a table column in the database. `RequiredAttribute` attribute is used for all string properties to override the database schema rule that allows a data field to be empty. The first relationship here is between the `PatientPersonalProfile` entity and the `Anamnesis` entity is 1:1. For this, a property of type `Anamnesis` is used in the `PatientPersonalProfile` class, and vice versa `Anamnesis` class will contain a property of type `PatientPersonalProfile`. The second relationship is 1:N between the `PatientPersonalProfile` entity and the `PresentDiagnose` entity; that is why the property `PresentDiagnoses` is defined as `ICollection` of type `PresentDiagnose`.

```
public class PatientPersonalProfile
    {
        [Required]
        public string PatientId { get; set; }
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        [Required]
        public DateTime DateOfBirht { get; set; }
        [Required]
        public string Gender { get; set; }
        [Required]
        [EmailAddress]
        public string Email { get; set; }
        [Required]
        public string PhoneNumber { get; set; }
        [Required]
        public string InsuranceNumber { get; set; }
        [Required]
        public string InsuranceKod { get; set; }
        public Anamnesis Anamnesis { get; set; }
        public ICollection<PresentDiagnose> PresentDiagnoses { get; set; }

    }
```

Source code 1: PatientPersonalProfile model class

The model classes are built in a separate project to improve the code's maintainability and readability. To be able to use these classes, `QRcare.Models` project must be referenced in `QRcare.API`.

The next step is to add database support on the server-side; this is done by creating a class that inherits from the EF built-in class `DbContext`, which is used to retrieve and manipulate the data in the database. *Source code 2* includes the part of `QRcareDbContext` class handling the `PatientPersonalProfile` entity only since this is a 130-line class. For that, we first define a `DpSet` property of `PatientPersonalProfile`, and by overriding the `OnModelCreating` method, we can configure relationships and properties to be the key to an entity. Also, we can provide seed data for the database as an initial set of data. Note here that the `QRcareDbContext` class includes an instance of `DbContextOptions`; this instance handles the configuration for the database and its connections.

```
public class QRcareDbContext : DbContext
    {
        public QRcareDbContext(DbContextOptions<QRcareDbContext> options)
            :base(options)
        {
        }
        public DbSet<PatientPersonalProfile> PatientPersonalProfiles { get; set;}
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<PatientPersonalProfile>()
                .HasKey(p => p.PatientId);
            modelBuilder.Entity<PatientPersonalProfile>()
               .HasOne(p=>p.Anamnesis)
               .WithOne(a => a.PatientPersonalProfile)
```

```
                .HasForeignKey<Anamnesis>(p=>p.PatientAnamnesisFK);
             modelBuilder.Entity<PatientPersonalProfile>()
                .HasMany(p => p.PresentDiagnoses)
                .WithOne(a => a.PatientPersonalProfile)
                .HasForeignKey(p=>p.PatientPresentFK);
             modelBuilder.Entity<PatientPersonalProfile>()
                 .HasData(new PatientPersonalProfile
             {
                 PatientId = "100",
                 FirstName = "Mohammad",
                 LastName = "Zabadi",
                 DateOfBirht = new DateTime(1995, 8, 16),
                 Gender = "Male",
                 Email = "zabadi@gmail.com",
                 PhoneNumber = "+470775421255",
                 InsuranceNumber="3015558077",
                 InsuranceKod="777",
             });
         }
     }
```
Source code 2: QRcareDbContext class - partial

The database connection string must be included in the `appsetting.json` file, and this is done as in *source code 3*.

```
"ConnectionStrings": {
        "DBConnection":
"Server=(localdb)\\mssqllocaldb;Database=QRcareDB;Trusted_Connection=True"
    },
```
Source code 3: Configuring database connection in appsetting.json file

Next is the `startup.cs` class, it must be modified as in *source code 4* for SQL Server configuration.

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<QRcareDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DBConnection")));
        }
```
Source code 4: Configuring database connection in startup.cs class

The last thing to do is to add a migration, and EF Core does this by instructing it using the two commands in *PowerShell commands 1*. `Add-Migration` command will create a directory called Migrations in the project with the migration class in it. It will also generate other configuration files. The `Update-Database` command will have EF Core create the database based on the schema from the migration.

```
Add-Migration InitialQRcare
Update-Database
```
PowerShell commands 1: EF commands for creating/adding migration and creating/updating database

By applying the previous steps, the application is now ready to run and operate on the database without the need to write any line of SQL, all handled by EF Core.

Whenever the models are modified, it will be enough to apply the same commands in *PowerShell commands 1* with a different migration name to apply the changes to the database.

4.4.1.2 Repository layer

Repository pattern intends to create an abstraction layer between the data access layer and the controller layer of an application. The repository separates the code responsible for database operations from the controller layer. This is achieved by building a repository interface for each entity and a repository class that implements the interface. *Source codes 5 and 6* are examples of repository implementation.

```
public interface IPresentDiagnoseRep
    {
        Task<IEnumerable<PresentDiagnose>> GetPresentDiagnoses(string patientId);
        Task CreatePresentDiagnose(PresentDiagnose presentDiagnose);
    }
```

Source code 5: IPresentDiagnoseRep interface for implementing the repository layer

This code declares the set of CRUD methods in the `PresentDiagnose` repository, one that returns all diagnoses for a single patient and one that creates a new diagnosis.

```
public class PresentDiagnoseRep : IPresentDiagnoseRep
    {
        private readonly QRcareDbContext qrcareDbContext;
        public PresentDiagnoseRep(QRcareDbContext qrcareDbContext)
        {
            this.qrcareDbContext = qrcareDbContext;
        }
        public async Task<IEnumerable<PresentDiagnose>>
        GetPresentDiagnoses(string patientId)
        {
            IQueryable<PresentDiagnose>
            diagnoses =qrcareDbContext.PresentDiagnoses;
            if(!string.IsNullOrEmpty(patientId))
            {
                diagnoses = diagnoses.Where
                (e => e.PatientPresentFK == patientId);
            }
            return await diagnoses.ToListAsync();
        }
    }
```

Source code 6: PresentDiagnoseRep repository class derived from the corresponding repository interface - partial

In the repository class, we first implement the corresponding interface, then the created DbContext class is injected through a constructor, and last, all CRUD methods are implemented. In *Source code 6*, only one of the one method will be shown.

4.4.1.3 Controller layer

This layer is responsible for responding to requests made against an ASP.NET Core MVC server. A controller is a C# class that derives from a base controller class, allowing it to inherit several useful methods. Controller actions are methods in the controller class that gets called when a particular URL is entered in the browser. (36) *Source code 7* contains part of the PresentDiagnoseController class.

```
[ApiController]
[Route("/diagnose")]
public class PresentDiagnoseController : ControllerBase
{
    private readonly IPresentDiagnoseRep presentDiagnoseRep;
    public PresentDiagnoseController(IPresentDiagnoseRep presentDiagnoseRep)
    {
        this.presentDiagnoseRep = presentDiagnoseRep;
    }
    [HttpGet("{patientId}")]
    public async Task<ActionResult<IEnumerable<PresentDiagnose>>>
    GetPatientDiagnoses (string patientId)
    {
        try
        {
            var result
            = await presentDiagnoseRep.GetPresentDiagnoses(patientId);
            if (result.Any())
            {
                return Ok(result);
            }
            return NotFound();
        }
        catch (Exception)
        {
            return BadRequest();
        }
    }
}
```
Source code 7: PresentDiagnoseController class for implementing the controller layer - partial

This Controller class is derived from the base class ControllerBase. Note that the correspondent repository is injected through a constructor. [ApiController] attribute indicates that the controller class is used to serve HTTP API responses, while [Route] attribute specifies the URL pattern. The method implemented here is a GET method, and the action here must be identified using the attribute `[HttpGet]`. Note that `{patientId}` is not a string but a reference to the variable `patientId`.

4.4.1.4 Access to patient's data

The section will describe the implementation of the server-side role in passing the permission to access the patient data from a patient to medical personnel. As described in the communication model in section 4.3.2, the mobile application will send an HTTP request for a key, and medical personnel will Send an HTTP request for the patient data using that key. The server-side is responsible for processing and responding to these two requests.

Processing key request

The first step for passing access permission is sending a request for a key from the mobile application to the server-side. The mobile application sends a POST request with the username and password of the patient to a specified URL, the server-side upon receiving the request, checks the user login received details if they match those on the database, if it passes, the server-side generates the key and saves it in the database for the user Id received in the request. The key generated is then serialized and sent back to the mobile application as a response. *Source code 8* illustrates this process.

```
public async Task<string> GetQRcodeKey(UserLogin user)
        {
            var checkPassword = await qrcareDbContext.UserLogins.
                                FirstOrDefaultAsync
                                (e => e.UserName == user.UserName);
            if (checkPassword is not null
                && checkPassword.Password == user.Password)
            {
                var result = await qrcareDbContext.QRCodeKeys.FirstOrDefaultAsync
                                   (e => e.PatientKeyId == user.UserId);
                if (result != null)
                {
                    string key = QRKeyGenerator.GetUniqueKey();
                    result.QrKey = key;
                    await qrcareDbContext.SaveChangesAsync();
                    string jsonKey = JsonSerializer.Serialize(key);
                    return jsonKey;
                }
                else
                    return null;
            }
            return null;
        }
```
Source code 8: GetQRcodeKey method in QRKeyRep class that checks for authentication and generate a digital key

The code above is from the `QRKeyRep.cs` in the repository folder. Note that this code calls for the method `GetUniqueKey` in `QRKeyGenerator.cs`. This method is responsible for generating a unique key every time it is called, meaning that this key is an on-request, one-time-use key, leading to this approach to be more secure. *Source code 9* is the code responsible for generating the key.

```
public class QRKeyGenerator
    {
        internal static readonly char[] chars =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890".ToCharArray();
        public static string GetUniqueKey()
        {
            byte[] data = new byte[80];
            using (var crypto = RandomNumberGenerator.Create())
            {
                crypto.GetBytes(data);
            }
            StringBuilder result = new StringBuilder(20);
            for (int i = 0; i < 20; i++)
            {
                var rnd = BitConverter.ToUInt32(data, i * 4);
                var idx = rnd % chars.Length;
                result.Append(chars[idx]);
            }
            return result.ToString();
        }
    }
```

Source code 9: QRKeyGenerator class responsible for generating the digital key

Processing patient's data request

Upon receiving the key, the patient can permit medical personnel to access the data simply by passing the key to the medical personnel in a QR code form. This will be illustrated later when describing the mobile application implementation. The medical personnel using the client-side sends the key to the server and requests the patient data; the server-side then looks for the key in the database and sends back the patient Id holding that key.

After receiving the patient Id on the client-side, it navigates to the patient page with the received Id as a route parameter in the URL, which leads to the client-side sending an HTTP GET request to the server for the patient data. The controller receives the request and passes it to the repository, where it processes it as illustrated in *Source code 10*.

```
public async Task<PatientPersonalProfile> GetPatient(string patientId)
        {
            var key = qrcareDbContext.QRCodeKeys.FirstOrDefault
                      (e => e.PatientKeyId == patientId);
            if (key.QrKey is not null)
            {
                key.QrKey = null;
                await qrcareDbContext.SaveChangesAsync();
                return await qrcareDbContext.PatientPersonalProfiles
                             .FirstOrDefaultAsync(e => e.PatientId == patientId);
            }
            return null;
        }
```

Source code 10: GetPatient method in PatientRep class that returns patient personal data after checking for a key

As seen above, the server first checks if there is a key stored with a `PatientKeyId` same as the `patientId` received in the request; if so, the server deletes the key, which emphasize that the key is a one-time-key, allowing only one access per key. The system then checks for the patient profile with the patient Id received in the request and sends it back to the client-side as a response.

Above was the implementation of the server-side of the application; the source codes captioned are just part of the whole server-side source code, the rest of the source code is similar in the structure, only differs in the way of data processing. The rest of the source code will be attached to the thesis.

4.4.2 Mobile application

The mobile application side of the project is implemented in Xamarin; this part of the solution consists of three different projects. The first project is the shared project; the author names it `QRcare`, where the collection of classes, models, services, views, and view models are located. This project allows sharing code across both `.Android` and `.iOS` projects, most of the application logic and coding will be implemented in this project. The other two projects are `QRcare.Android` and `QRcare.iOS`, they are named automatically by VS after the first project. Android and iOS applications are built differently, so before deploying the mobile application on either of them, some configurations must be done, which is the purpose of `QRcare.Android` and `QRcare.iOS` projects.

The application architecture is built in an MVVM architectural pattern; therefore, the application will be split into models, views, and view models. The implementation of the MVVM pattern will be discussed in the following sections.

4.4.2.1 Models

This is the part where business logic and data objects are implemented, and it also covers the interaction with web services for data retrieval and on-device data storing.

Same as the server-side, the data in the mobile application is modelled as classes with properties; these models will serve as data mapping from the server to the mobile application. This application will contain three models: `QRKeyModel` for key mapping, `UserLogin` for login data mapping, and `UserModel` for patient data mapping. This work will not include the source code for the models.

The service layer is included in this part of the application; this layer is responsible for requesting data from the server-side; the requested data is then cashed and mapped to the UI elements through view models.

The REST web requests are implemented in one class called `QRKeyService` in the Service directory. In this class, we first need to specify the Web API URL, where the request will be sent, followed by implementing CRUD operations. Note that the data sent to the server must be serialized and encoded to an HTTP request format, then deserialize the data in the HTTP response. *Source code 11* illustrates configuring the web service layer and requesting the patient personal data through a GET request.

```
public static class QRKeyService
    {
        static string BaseUrl = "https://qrcareapi.azurewebsites.net/";

        static HttpClient client;

        static QRKeyService()
        {
            client = new HttpClient
            {
                BaseAddress = new Uri(BaseUrl)
            };
        }
        public static async Task<UserModel>
        GetPatientProfile(UserLogin userLogin)
        {
            var json = JsonConvert.SerializeObject(userLogin);
            var content = new StringContent
            (json, Encoding.UTF8, "application/json");
            var response = await client.PostAsync("/mobile", content);
            JObject jsonresponse = JObject.Parse
            (response.Content.ReadAsStringAsync().Result);
            JToken token = jsonresponse;
            UserModel user = token.ToObject<UserModel>();
            return user;
        }
    }
```
Source code 11: QRKeyService class, the implementation of web service layer and GetPatientProfile method - partial

4.4.2.2 View Models

View models contain the code behind the views; it is separated from the views to improve the readability and ease the unit testing. View models implement the properties and commands to which the view can bind data to. It is also responsible for coordinating the data between models and views. View model receives data and events from views based on user interaction, then it retrieves the data from models and sends it back to the views in a form it can easily consume. (37)

View models are connected to the views through XAML or the view's code-behind file. The second is used by the author as illustrated in *source code 12*.

```
public partial class HomePage : ContentPage
    {
        public HomePage()
        {
            InitializeComponent();
            this.BindingContext = new HomeViewModel();
        }
    }
```

Source code 12: the code-behind of the HomePage view

Source code 13 is a partial code implementation of the `HomeViewModel`; this view model is responsible for authenticating the patient before requesting a key from the server.

```
class HomeViewModel : BaseViewModel
    {
        public Command GeneretaCommand { get; }
        public Command GeneretaFingerprintCommand { get; }
        private string password;

        public HomeViewModel()
        {
            Title = "QRcare";
            GeneretaCommand = new Command(OnGenerateClicked);
            GeneretaFingerprintCommand
            = new Command(onGeneretaFingerprintCommand);
        }
        public string Password
        {
            get { return password; }
            set
            {
                SetProperty(ref password, value);
            }
        }
        private async void OnGenerateClicked(object obj)
        {
            var isAuth = await QRKeyService.Auth
            (Preferences.Get("Username", string.Empty), Password);
            if (isAuth != null)
            {
                await Shell.Current.GoToAsync($"//{nameof(QRCodePage)}");
            }
            else
            {
                await Application.Current.MainPage.DisplayAlert
                ("Unauthorized", "Wrong Password, please try again.", "OK");
            }
        }
    }
```

Source code 13: HomeViewModel class - partial

The password property and one command are implemented in the above source code; the complete source implements two commands, but only one command is covered in this work. The implementation of `GeneretaFingerprintCommand` will be available in the complete source code attached to this work.

The password property is one-way bound to an entry element on the `HomePage` view, as it will be illustrated later in *Source code 14*. Pressing Generate button in the view will fir an event that triggers `GenereteCommand`, where the view model authenticates the user by the password entered. When the user is authenticated, the shell will display the `QRCodePage` view. Otherwise, it will display a dialog box with the alert provided above.

4.4.2.3 Views

This part is responsible for displaying the information to the user; this is where the user interface is defined and implemented using XAML. Views are also responsible for data binding and sending events to the view models. Events are triggered by user interaction like pressing a button; or self-triggered like setting a timer. When the user presses a button, this fires an event that is sent to the view models, where a chain of actions is performed. Using Two-Way binding, views can send data to view models and vice versa. (37) an example of this is *source code 14*, which is a partial XAML from the `HomePage` view.

```
<StackLayout Orientation="Vertical" Padding="20" Spacing="20">
                <Label Text="Generate QR code using Password or Fingerprint"
                       TextColor="Black"
                       FontSize="Large" HorizontalTextAlignment="Center"/>
                <Label Text="Password" Padding="0,20,0,0" TextColor="Black"/>
                <Entry IsPassword="True"
                       Text="{Binding Password}"
                       TextColor="Black"
                       BackgroundColor="{StaticResource Accent}"/>
                <StackLayout BackgroundColor="{StaticResource Primary}"
                             Orientation="Horizontal">
                    <Button VerticalOptions="Center"
                            ImageSource="icon_QR.png" Text="Generate"
```

```
                              Command="{Binding GeneretaCommand}"
                              HorizontalOptions="StartAndExpand"
                              BackgroundColor="{StaticResource Primary}" />
                      <Button VerticalOptions="Center"
                              ImageSource="icon_fingerprintlogin.png"
                              Text="fingerprint"
                              Command="{Binding GeneretaFingerprintCommand}"
                              HorizontalOptions="EndAndExpand"
                              BackgroundColor="{StaticResource Primary}" />
                  </StackLayout>
              </StackLayout>
```

Source code 14: XAML markup of the Home View - partial

Note in this example that using a One-Way binding, the element `<Entry>` is bound to the property Password in `HomeViewModel`. When the user enters a value here, the property gets updated in real-time. The element `<Button>` is bound to the `GenerateCommand` command; pressing this button will trigger this command on the view model. The properties here define the design of XAML elements. For instance, the horizontal position of the element is defined using the `HorizontalOptions` property. Elements like `<StackLayout>` define the layout of the view.

4.4.2.4 Requesting a key

The primary purpose of the mobile application is to allow the patient to grant medical personnel access to the patient's data. Using the mobile app, the patient sends a request to the server for a key displayed on the mobile as a QR code. This section will go through the actions performed by the mobile application for requesting the key.

When the mobile application is launched, the patient will be asked to log in using a password or to be authenticated using a fingerprint. For the fingerprint authentication to be available, the patient must use the correct password the first time he logs in, and the password is then stored in the application to be used later for fingerprint authentication. This part implementation will not be covered in this work.

When the user is authenticated, the application will display the `HomePage` view where the patient can request a key; the patient here will be authenticated again using a password or fingerprint before requesting a key. This is illustrated in *Source codes 13 and 14*.

When the user is authenticated, the `QRCodePage` view is displayed, the view model of this view will send an API request to the server for requesting a key; if the request is successful, the key is sent back to view, where it will be displayed as QR code. *Source code 15* is part of the view model code.

```
public async void LoadQRKey()
    {
        Key = await QRKeyService.GetQRKey(DataStore.GetUser().Result);
    }
public void OnAppearing()
    {
        LoadQRKey();
    }
private async void OnCloseClicked()
    {
      await QRKeyService.DeleteQRKey(DataStore.GetUserId().Result);
      await Shell.Current.GoToAsync($"//{nameof(HomePage)}");
    }
```

Source code 15: QRCodeViewModel code responsible for requesting, data binding, and deleting the key

The `OnAppearing()` method is overridden in the code-behind of the view, this method will be triggered when the view appears on the shell, where it calls `LoadQRKey()` method that calls `GetQRKey` method in `QRKeyService` class, `GetQRKey` method implementation is shown in *Source code 16*, note that this method expects a `UserLogin` as a parameter, which will be provided by the data stored in the application. The `OnCloseClicked` method is triggered by the `CloseCommand` that is bound to the close button, and this method will then send a DELETE request to the server to delete the key from the database.

```
public static async Task<string> GetQRKey(UserLogin user)
        {
            var json = JsonConvert.SerializeObject(user);
            var content = new StringContent
                          (json, Encoding.UTF8, "application/json");
            var response = await client.PostAsync("/QRKey", content);
            if (response.IsSuccessStatusCode)
            {
                var result = await response.Content.ReadAsStringAsync();
                return JsonConvert.DeserializeObject<string>(result);
            }
            return null;
        }
```

Source code 16: GetQRKey method in QRKeyService class

Here, the `GetQRKey` method serializes the `UserLogin` parameter passed, encodes it, and send it as an HTTP POST request to the server, the response which is the string key is then deserialized and returned to the view model that passes it using data binding to the view, the view then transforms it to a QR code as shown in *Source code 17*.

```
<zxing:ZXingBarcodeImageView BarcodeValue="{Binding Key}"
                                     HeightRequest="300"
```

```
                                        WidthRequest="300"
                                        HorizontalOptions="CenterAndExpand"
                                        VerticalOptions="CenterAndExpand">
                    <zxing:ZXingBarcodeImageView.BarcodeOptions>
                        <zxingcommon:EncodingOptions Height="300" Width="300" />
                    </zxing:ZXingBarcodeImageView.BarcodeOptions>
```
Source code 17: implementing ZXing through XAML for generating the QR code

ZXing is the NuGet package that allows transforming a text (the binding key in this case) into a QR code. This view also implements a Cancel button that sends an HTTP DELETE request to the server to delete the key from the database.

This section concludes the implementation of the mobile application. It is essential to mention that this application is configured to run on Android but not iOS since iOS applications can only be configured and tested in macOS. However, Windows OS was used during the implementation of the system.

4.4.3 Client-side

The client-side of the application will run on the browser, and it is the side of the system that will be used by medical personnel to access, view, and manipulate the patient's data. The client-side will be implemented in Blazor WebAssembly, chosen over Blazor Server for its advantages, mainly its low latency compared to Blazor Server.

4.4.3.1 Project structure

Blazor WebAssembly's initial structure is defined on creating the project template, and the structure will be described starting from the project's entry point and continuing with the app run life cycle. (38)

- `Program.cs:` This file is the app's entry point responsible for setting up the WebAssembly host. The root component (`App.Razor`) for the app is defined here.
- `Wwwroot:` This folder contains the public static assets of the app, such as CSS files and the app icon. But most importantly, `index.html` file, which is the root page of the app, this HTML file defines where the root component is rendered by specifying an `<div>` element with the id 'app.'
- `App.razor:` This is the app's root component that sets up client-side routing using the built-in Router component. The root component also defines the layout component, by default, its `MainLayout.razor`.

- `MainLayout.razor:` The application's main layout component, here the navigation, main, and footer are defined. This component is placed in the Shared folder alongside other components, such as `NavMenu`.
- `NavMenu.razor`: This component implements the sidebar navigation. The `NavLink` component in `NavMenu` is responsible for rendering navigation links to other Razor components.
- `_Import.razor:` This is not a Razor component file; this file defines the common Razor directives to import into the app's components, such as `@using` directives for namespaces.
- Pages folder: This folder contains the routable pages (which are components with `.razor` extension) rendered in the Blazor app's body. These pages are defined by assigning a route for them using the directive `@page`.
- `Components` folder: This folder contains the reusable components that can be rendered in other components. Unlike components in the pages folder, components here have no specific route.
- `Services` folder: contains the collection of classes and interfaces responsible for sending HTTP requests and receiving HTTP responses.

As mentioned before, Blazor apps are built using Razor components, consisting of Razor markup, HTML, and C#. C# code is written in `@code` block in Razor component. However, this code can be separated into a class using the base class approach; the component then specifies its base class using the `@inherits` directive. This can be noticed in the Pages folder where `PatientPage.razor` inherits `PatientPageBase.cs` class.

In the following sections, the process of scanning the QR key and navigating to the patient page will be described. The QR scan page will cover the process of scanning the QR key, while the patient page will cover the process of retrieving the patient data and how the HTTP request is sent to the server.

4.4.3.2 QR scan page

This page is responsible for scanning the QR code, which is the key given by the patients to access their data. Note that a QR scanner can be used for scanning the QR code.

However, a webcam scanner was implemented as part of an improved independent system, decreasing the need for external hardware for the system to function.

This page consists of an input text and two buttons, one that opens the webcam for scanning the QR key, and the other navigates to the patient page of the QR key holder. *Source code 18* is a snippet code of the Razor component markup.

```
<div class="field is-grouped">
    <div class="col-sm-11">
        <EditForm Model="@Key">
            <InputText id="PatientId" class="form-control"
                       plceholder="Patient ID" @bind-Value="Key" />
        </EditForm>
    </div>
    <p class="control">
       <a class="button">
          <span class="oi oi-camera-slr"
                aria-hidden="true" @onclick="OpenCamera"></span>
       </a>
    </p>
</div>
<div class="col-sm-10">
   <button class="button is-primary is-light"
           @onclick="ReadQRKey">Submit</button>
</div>
```

Source code 18: Razor markup part of QrScanPage page/component - partial

Note that `EditForm` and `InputText` are nested built-in Blazor components. `EditForm` is used mainly for validation, where the Model attribute specifies the instance to be validated. `InputText` is an input component for editing string value that binds it to a specified property using the `@bind-Value` directive. In the above source code, the `InputText` is bound to the Key property. Both components are helpful in handling events such as `onchange` or onclick events, specifying the actions to be performed on such events.

When the button with the camera icon is pressed, this triggers the `@onclick` event that calls the `OpenCamera` method since it is bound to it. The code part of the Razor components is placed in a base class. *Source code 19* is the component code.

```
public class QRScanPageBase : ComponentBase
    {
        [Inject]
        public IQRKeySer QRKey { get; set; }
        [Inject]
        NavigationManager NavigationManager { get; set; }
        public string Key { get; set; } = string.Empty;
        public async Task ReadQRKey()
        {
           string id=  await QRKey.GetPatientId(Key);
           NavigationManager.NavigateTo($"/patient/{id}");
        }
        public BarcodeReader Reader=new BarcodeReader();
        public void OpenCamera(MouseEventArgs args)
        {
            if (Reader.IsDecoding)
                Reader.StopDecoding();
            else
                Reader.StartDecoding();
        }
        public void LocalReceivedBarcodeText(BarcodeReceivedEventArgs args)
        {
            Key = args.BarcodeText;
            Reader.StopDecoding();
            Reader.Dispose();
            StateHasChanged();
        }
    }
```
Source code 19: Razor code part of QrScanPage page/component

Starting from the top, this class is derived from `ComponentBase` built-in class, allowing it to be inherited by the Razor component. `[Inject]` attribute allows us to access services using the dependency injection technique. In this example, we are injecting `IQRKeySer,` which is an HTTP service, and `NavigationManager` for managing URL navigation. Next, we have the property '`Key`' bound to an `InputText` as shown in *Source code 18*. The object Reader is an instance of a class from an installed ZXing NuGet package responsible for scanning the QR code.

To read the code, the user must press the camera-icon button to access the webcam that acts as a scanner here; this calls the method '`OpenCamera`,' which checks if the webcam is on and reverses its status.

```
<BlazorBarcodeScanner.ZXing.JS.BarcodeReader Title=""
                                             @ref="Reader"
                                             StartCameraAutomatically="false"
                                             ShowStart="false"
                                             ShowReset="false"
                                             ShowToggleTorch="false"
                                             ShowVideoDeviceList="false"
                                             VideoWidth="3000"
                                             OnBarcodeReceived=
                                             "LocalReceivedBarcodeText"
                                             VideoHeight="2000"
                                             ShowResult="false"/>
```
Source code 20: Nesting ZXing component in QrScanPage page/component

Source code 20 is the code nesting the ZXing component responsible for accessing the webcam and scanning the QR code. Note that the `@ref` directive references this component to the defined object Reader. When the camera successfully scans the code and receives a value, it calls the `LocalRecivedBarcodeText` method, assigning the value received to the property '`Key`.'

Finally, the submit button is bound to the method `ReadQRKey`, pressing it will call the method that sends a GET request with the key as a parameter, the server checks for the key in the database and responses with the patient id having the key, then on the client-side, the component navigates to the patient page of the route "`./patient/{id}`," where `{id}` is a variable, not a string. This is illustrated in *Source code 19* above.

4.4.3.3 Patient page

This section will use the patient page as an example of how Blazor is implemented and how the app sends HTTP requests to the server.

The patient page is responsible for displaying the patient medical data retrieved from the server, allowing authenticated medical personnel to add to/update the data retrieved and send the changes back to the server to be saved on the database.

For that, the patient page will implement the functionality for displaying the patient personal data, displaying diagnoses with the ability to add new diagnoses, and displaying the anamnesis with the ability to update them.

`PatientPage.razor` file is a Razor component itself, meaning that it is built on Razor syntax, consisting of Razor markup, HTML, and C#, the C# code is placed in a base class. Using component nesting, we can include a component into another component, and the nested components are declared using HTML syntax, where the component name is used as an HTML tag name. *Source code 21* is a snippet code from `PatientPage.razor`.

```
@page "/patient/{PatientId}"
@inherits PatientPageBase

@if (PatientPersonalProfile is null)
{
    <SfSpinner Visible="true" Label="Loading"
               Type="@SpinnerType.Fabric" Size="100"></SfSpinner>
}
else
{
    <div class="panel">
        <div class="patient-section__heading text-nowrap">
            <label class="panel-title">Patient</label>
        </div>
    </div>
    <div class="form-group ">
```

```
        <label class="col-sm-2 col-form-label">
            First Name
        </label>
        <label class="col-sm-2 col-form-label">
            @PatientPersonalProfile.FirstName
        </label>
    </div>
}
```

Source code 21: Razor markup part of PatientPage page/component - partial

After scanning the QR key, the client-side navigates to the patient page with the patient Id as the route parameter; the route parameter is used to populate the corresponding component parameter with the same name. Note the `@page` directive uses `{PatientId}` as a route parameter that assigns the value of the route segment to the property `PatientId` having the `[Parameter]` attribute as shown in *Source code 22*. (39)

```
[Parameter]
public string PatientId { get; set; }
protected async override Task OnInitializedAsync()
{
   PatientPersonalProfile = await PatientSer.GetPatient(PatientId);
   if (PatientPersonalProfile is null)
   {
       NavigationManager.NavigateTo("404");
   }
   else
   {
       PresentDiagnoses =(await PresentDiagnoseSer.GetPresentDiagnoses
                              (PatientId)).ToList();
       Anamnesis = await AnamnesisSer.GetAnamnesis(PatientId);
    }
}
```

Source code 22: Razor code part of PatientPage page/component - partial

The `PatientId` property is then used to send an HTTP GET request for patient data, including personal profile, diagnoses, and anamnesis. This HTTP request is sent when the page is called using the overridden method `OnInitializedAsync()`. *Source code 23* is the implementation of the HTTP service responsible for sending the HTTP request and receiving the HTTP response.

```
public class PatientSer : IPatientSer
    {
        private readonly HttpClient httpClient;
        public PatientSer(HttpClient httpClient)
        {
            this.httpClient = httpClient;
        }

        public async Task<PatientPersonalProfile>
                     GetPatient(string patientId)
        {
           return await httpClient.GetFromJsonAsync<PatientPersonalProfile>
```

```
                              ($"patient/{patientId}");
          }
     }
```
Source code 23: implementation of HTTP service for patient personal data

The patient page also allows the medical personnel to review the old diagnoses, add a new one, and review and update the anamnesis. However, the implementation of those will not be covered in this work.

The event of adding a new diagnosis or updating the anamnesis causes the component to be rendered again without the need to refresh the page. In other words, the medical personnel will still have access to manipulate the patient data as long as he does not navigate out of the patient page. Once he does, the session is closed, and all access to patient data will be denied until the patient allows it by requesting a key again.

With that, we conclude the implementation phase of the three sides of the system, noting that several functionalities of the system that are implemented were not covered in this work. However, the implementation of these functionalities follows the general concept of implementation that was described.

4.5 Testing

The final phase of the practical part is testing the software implemented. Typically, several software tests are performed on the software before determining whether it is ready to be used by public users. However, only system testing and usability testing will be performed on the implemented healthcare system.

4.5.1 System testing

This level of testing validates the integrated software application as a whole. The purpose of this test is to evaluate the system's compliance with its specified requirements. The author performed system testing during the implementation phase of the system upon implementing each of the specified requirements in the analysis phase individually, if the system performs the tasks (like scanning the QR code), retrieves the data (like displays the correct patient data), and saves the changes of data on the database (like saving the new added diagnose) correctly, the system passes the test. Otherwise, the system is debugged to find the reason for fixing such an error.

After finishing the system implementation, the system was published on Microsoft Azure to perform usability testing. The author applied the configuration needed for the system's sides to communicate with each other over the internet instead of the local host network; the author then deployed the system on Microsoft Azure and performed the system test for one last time to validate all the system functionalities.

4.5.2 Usability testing

Users perform this system evaluation to evaluate how user-friendly the software is by asking participants to perform multiple tasks while observing them and listening to their feedback. (40) For the implemented system, 4 participants were asked to evaluate the system usability by performing the following tasks:

- Log in to the mobile application using the provided user login data.
- Request a key using the mobile application.
- Scan the key using the Blazor WebAssembly application on the browser.
- Review the patient's medical data.
- Manipulate the patient's data by adding new diagnoses and updating the anamnesis.

After performing these tasks, the participants were asked for feedback on the tasks performed. They were also asked about likes, dislikes, and suggestions to improve the system.

5 Results and discussion

5.1 Test results

The participants were asked to perform the five main tasks of the system. The first two tasks were performed using the mobile application, where all 4 participants could finish the given tasks successfully. The only issue they pointed out was that the application does not show a loading screen indicating that the system is performing the login process; it takes about 6 seconds for the system to perform this process. Also, 1 of the 4 participants did not like the colours used in the application.

As for the next three tasks, the participants used the client-side (the Blazor application) to perform them. The application starts at the login page, which is implemented with a code-behind. However, the participants were informed that the login in page is not functional, and pressing the login button is sufficient to navigate to the QR scan page for the participants to start performing the tasks.

3 out of the 4 participants managed to find out immediately how to start the webcam to scan the QR code. One participant pressed submit button on the first try and was directed to the "404" page. However, he succeeded on the second try.

All participants recognized the data displayed on the patient page and displayed the additional information for a specific diagnosis. However, only two of them noticed that the anamnesis section was collapsible and pressing the Anamnesis panel will show the patient's anamnesis.

All participants successfully added new diagnoses and successfully updated the anamnesis.

When the participants were asked for feedback on the client-side application, all participants gave positive feedback regarding error prevention and visibility of system status and that the system shows whether the diagnoses and anamnesis are updated or not.

Some suggestions given by participants to improve the system are:

- Implement a loading indicator for the login view of the mobile application.
- Display the patient personnel data on the patient page in a more readable and user-friendly way.

- Implement a search function for the diagnoses section to ease finding the related diagnoses when the patient has many.
- Implement a confirmation dialog where the user must confirm to apply the changes performed.

5.2 Discussion

5.2.1 System usability

The system's main users are patients (the public users) and medical personnel. The usability testing is focused on the system areas used by public users, which are the mobile application, and the way the data is displayed on the patient page (this is a part of further system improvements allowing patients to login using the client-side and review their own data), the system passed the usability testing covering these areas. As for the medical personnel, special training will be provided to clarify the system functionalities. The implemented prototype is not so complex to require training to be used, but if the system is to be further improved, the complexity will grow, and the medical personnel must be trained to use it.

5.2.2 Proposed improvements

This work is only a prototype of a real-world healthcare system; this prototype can be further improved and developed to be used by the public. In the short run, the author is proposing the following improvements:

- Implementation of authentication and authorization on the client-side, allowing only authenticated users to use it, and based on the user role in the system, only the needed data and functionalities will be available.
- Extending the database and client-side to include other needed medical data (like treatments) and other health institutions such as dentistry and pharmacy.
- Improving the mobile application and adding functionalities such as a calendar for appointments and a notifications system for notifying the patient of various events, such as a treatment reminder and notifying the patient when the data is accessed or changed.
- Styling improvements on both client-side and mobile applications.

- Security improvements on the server-side.

5.2.3 Benefits of the system

In addition to the benefits brought by WebAssembly and Blazor WebAssembly that were described in the literature overview, the implemented system brings major benefits for both developers and the medical section. First, for developers, being able to work with .NET environment and code in C# gives a developer a full-stack experience. It allows them to implement native and cross-platform applications, especially with the .NET new framework MAUI that allows developing apps to run on both mobile and desktop using a single shared codebase. Simply put, with Blazor WebAssembly and MAUI, an application can run on more than %96 of all mobile and desktop devices.

On the other hand, the medical section and the healthcare institutions can benefit from the system by providing a shared platform for all medical personnel, where all the patient's medical record data are saved to and retrieved from one place securely and efficiently and allows a patient to grant health institutions access to the patient's medical data in the simplest and most secure way.

Another benefit is that a digital healthcare system allows automation to take place in the medical section by using artificial intelligence and neural network in the healthcare system, which will help detect and reduce medical mistakes caused by the human factor, such as prescription errors, and alerting medical personnel on error occurrence. Such a system also helps detect genetic diseases and keep track of a patient's health condition. In conclusion, a digital healthcare system benefits both patients and the medical section.

6 Conclusion

The goal of the thesis was to develop and implement a healthcare system in the .NET platform and define the technologies, frameworks, platforms within .NET needed to implement the system. This was achieved by implementing a system that consisted of a server-side, client-side, and mobile application. The system sides were implemented in ASP.NET Core, Blazor WebAssembly, and Xamarin. The system also includes a database built in SQL Server. Other technologies, frameworks, and patterns were used to build a durable system, such as Entity Framework, MVVM, MVC, and more. The theoretical part of the thesis describes all the concepts used during the system implementation. The discussion section of this thesis discusses some advantages of the presented prototype and proposed improvements that can be applied to the system.

Another goal was to introduce WebAssembly and Blazor WebAssembly technologies, which was achieved in the theoretical part discussing both technologies in detail alongside Blazor Server to differentiate it from Blazor WebAssembly.

This work proposes an approach for managing data access permission, where a patient can grant permission to access the patient's data by passing a digital key in the form of a QR code generated by the server. This key is a one-time on-request key, meaning that a new key is generated on the patient request and is deleted after it is used to access data.

7 Bibliography

1. Usage Statistics of JavaScript as Client-side Programming Language on Websites, February 2022. *W3Techs - extensive and reliable web technology surveys.* [Online] 2 12, 2022. https://w3techs.com/technologies/details/cp-javascript/.

2. Himschoot, Peter. *Microsoft Blazor: Building Web Applications in .NET.* Melle, Belgium : Apress, 2020. 9781484259276.

3. Engström, Jimmy. *Web Development with Blazor and .NET: A hands-on guide to building interactive web UIs with Blazor and C#.* Birmingham, UK : Packt Publishing Ltd., June 2021. 978-1-80020-872-8.

4. WebAssembly. *WebAssembly.* [Online] https://webassembly.org/.

5. WebAssembly Support. *caniuse.* [Online] 29 1 2022. https://caniuse.com/wasm.

6. FAQ - WebAssembly. *WebAssembly.* [Online] https://webassembly.org/docs/faq/.

7. Introduction to ASP.NET Core Blazor. *Microsoft Docs.* [Online] https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0.

8. ASP.NET Core Razor components. *Microsoft Docs.* [Online] Microsoft. https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-6.0.

9. Wright, Toi B. *Blazor WebAssembly by Example: A project-based guide to building web apps with .NET, Blazor WebAssembly, and C#.* Birmingham UK : Packt Publishing, 2021. 9781800567511.

10. Introduction to ASP.NET Core SignalR. *Microsoft Docs.* [Online] https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-6.0.

11. ASP.NET Core Blazor hosting models. *Microsoft Docs.* [Online] https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0.

12. Daniel Roth, Rick Anderson, Shaun Luttin. Overview of ASP.NET Core. *Microsoft Docs.* [Online] Microsoft , 3 4 2022. https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0.

13. LOCK, ANDREW. *ASP.NET Core in Action.* Shelter Island, NY : Manning Publications, 2018. 9781617294617.

14. Lauret, Arnaud. *The Design of Web APIs.* s.l. : Manning, 2019. 9781617295102.

15. Larkin, Rick Anderson and Kirk. Create a web API with ASP.NET Core. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio.

16. Smith, Steve. Overview of ASP.NET Core MVC. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?WT.mc_id=dotnet-35129-website&view=aspnetcore-6.0.

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

18. Overview of Entity Framework Core - EF Core | Microsoft Docs. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/ef/core/.

19. Microsoft Docs. *Compare EF6 and EF Core | Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/ef/efcore-and-ef6/.

20. Migrations Overview - EF Core. *Microsoft Docs.* [Online] Microsoft, 27 10 2021. https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli.

21. SQL Server Tutorial. *What is SQL Server.* [Online] https://www.sqlservertutorial.net/getting-started/what-is-sql-server/.

22. What is Xamarin? - Xamarin. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin.

23. Xamarin.Essentials - Xamarin . *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/xamarin/essentials/.

24. What is Xamarin.Forms? - Xamarin | Microsoft Docs. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms.

25. Mazloumi, Dan Hermes and DR. Nima. *Building Xamarin.Forms Mobile Apps Using XAML.* s.l. : Apress, 2.2019. 9781484240304.

26. The Model-View-ViewModel Pattern - Xamarin . *Microsoft Docs.* [Online] Microsoft. https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

27. .NET (and .NET Core) - introduction and overview . *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/dotnet/core/introduction#net-core-net-framework-mono-uwp.

28. What's new in .NET 5 . *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-5.

29. Object-Oriented Programming (C#) . *Microsoft Docs.* [Online] Microsoft. https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop.

30. by Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde. *The C# Programming Language (3rd Edition).* s.l. : Addison-Wesley Professional, 2004. 978-0321562999.

31. A Tour of C# - C# Guide. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

32. David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, Sailu Reddy. *HTTP: The Definitive Guide: The Definitive Guide (Definitive Guides).* s.l. : O'Reilly Media, 2002. 978-1565925090.

33. GitHub - zxing/zxing: ZXing ("Zebra Crossing") barcode scanning library for Java, Android. *GitHub.* [Online] zxing. https://github.com/zxing/zxing.

34. Software Development Life Cycle (SDLC) - Big water Consulting. *Big water Consulting.* [Online] 8 4 2019. https://bigwater.consulting/2019/04/08/software-development-life-cycle-sdlc/.

35. ČR, síť lékařských fakult. Anamnéza – WikiSkripta. *WikiSkripta.* [Online] https://www.wikiskripta.eu/w/Anamn%C3%A9za.

36. Walther, Stephen. ASP.NET MVC Controller Overview (C#) . *Microsoft Docs.* [Online] Microsoft , 19 2 2020. https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs.

37. Johnson, Paul. *Using MVVM Light with your Xamarin Apps.* Merseyside, UK : Apress, 2018. 9781484224748.

38. ASP.NET Core Blazor project structure. *Microsoft Docs.* [Online] Microsoft. https://docs.microsoft.com/en-us/aspnet/core/blazor/project-structure?view=aspnetcore-6.0.

39. ASP.NET Core Blazor routing and navigation. *Microsoft Docs.* [Online] Microsoft . https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/routing?view=aspnetcore-6.0.

40. Petra Pavlíčková, Josef Pavlíček. Business Process Models (BPMN and DEMO Notation) - Usability Study. [book auth.] Eduard Babkin, Russell Lock, Pavel Malyzhenkov, and Vojtěch Merunka Robert Pergl. *Enterprise and Organizational Modeling and Simulation.* Prague, Czech Republic : Springer International Publishing AG, 2019.