

UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione



TESI DI LAUREA

**Progettazione e implementazione di un'app in tecnologia MAUI per
supportare i visitatori di un museo tattile**

**Design and implementation of an app in MAUI technology to
support visitors to a tactile museum**

Relatore

Prof. Domenico Ursino

Correlatore

Dott. Enrico Corradini

Candidato

Valerio Morelli

ANNO ACCADEMICO 2022-2023

*Great things are not done by impulse,
but by a series of small things
brought together.*

Vincent Van Gogh

Sommario

Negli ultimi anni, l'adozione dei framework multiplatforma ha registrato una crescente rilevanza nell'ambito dello sviluppo software. Questi strumenti, consentendo la creazione di applicazioni compatibili su diverse piattaforme, soddisfano la necessità di raggiungere un pubblico più ampio con dei tempi di sviluppo ridotti. Questa tesi propone i risultati di uno studio del framework .NET MAUI stabilitosi nel 2022 come successore di Xamarin. Verranno esplorati numerosi aspetti teorici la cui comprensione risulta necessaria alla creazione di un progetto con tale framework e verrà quindi proposta la realizzazione di un'applicazione gestionale per migliorare la qualità dei servizi offerti da un museo.

Keyword: MAUI, Microsoft, .NET, Xamarin, Cross-platform, Firebase, Software Gestionale, Museo Tattile Omero

Abstract

In recent times, there has been a growing significance of cross-platform frameworks in the field of software development. Such tools facilitate the development of applications that run seamlessly on different platforms and enable wider market share reach with less workload. This dissertation presents the findings of a study conducted on the .NET MAUI framework, which was introduced in 2022 as the successor to Xamarin, and represents Microsoft's attempt to broaden its position within this area. Numerous theoretical concepts fundamental to building a project with this framework will be investigated, and a management application will be developed to enhance the quality of services provided by a museum.

Keyword: MAUI, Microsoft, .NET, Xamarin, Cross-platform, Firebase, Management software, Omero Tactile Museum

Introduction	1
1 Introduction to MAUI technology	3
1.1 An Overview of the .NET World	3
1.1.1 What is .NET?	3
1.1.2 Compilation and runtime process	4
1.2 The birth of MAUI	4
1.2.1 Project structure	4
1.2.2 WinUI and MacCatalyst	5
1.2.3 A loosely-coupled approach	5
1.2.4 Dependency injection	7
1.2.5 Hot reload	7
1.2.6 MAUI's compilation process	7
1.2.7 MAUI's layered architecture	8
1.3 XAML	8
1.3.1 The limits of XAML	9
1.3.2 XAML markup extensions and shared resources	9
1.3.3 C# Markup	10
1.4 Architectural patterns	10
1.4.1 MVVM	12
1.4.2 MVVM Toolkit	13
1.4.3 MVU	14
1.5 Data binding	16
1.5.1 Communication between loosely coupled components	16
1.5.2 The structure of data binding	18
1.5.3 Binding mode	19
1.5.4 Binding value converters	19
1.5.5 Multi-binding	20
1.6 Wrapping up	20
2 The Omero Tactile Museum	21
2.1 Foundation	21
2.1.1 The idea for an accessible museum	21
2.1.2 The Omero Museum today	22
2.2 Internal organization	22

2.2.1	Services offered	23
2.2.2	How a client visit takes place	23
2.2.3	The need for management software	24
2.2.4	Internal activities analysis	25
3	Requirements Analysis	26
3.1	Application requirements	26
3.1.1	Functional requirements	26
3.1.2	Non-functional requirements	27
3.2	Actors and use cases	28
3.2.1	Actors	29
3.2.2	Use cases diagrams	30
3.3	Mapping matrix	35
3.4	Class diagram	36
3.4.1	Data requirements	36
3.4.2	Translation towards the object-oriented model	36
4	Software design	38
4.1	Database structure	38
4.1.1	Firebase Realtime Database	38
4.1.2	Nodes design	39
4.2	Applications screen design	39
4.2.1	Applications map	40
4.2.2	Mockups	41
4.3	Refined class diagram	42
4.3.1	The MVVM as the reference architectural pattern	42
4.3.2	Overall software structure	47
4.4	Sequence diagrams	48
4.5	Activity diagrams	49
5	Implementation	56
5.1	Development technologies	56
5.1.1	The MAUI framework	57
5.1.2	Firebase services	58
5.2	Views implementation	60
5.2.1	Supporting components	61
5.2.2	Applications' Shells	63
5.2.3	NuGet packages	64
5.3	Database seeders	67
5.4	Issues encountered	67
6	User's manual	69
6.1	Staff application usage	69
6.1.1	Installation and setup	69
6.1.2	Authentication	70
6.1.3	Core Functionalities	71
6.2	Customer application usage	73
6.2.1	Installation and setup	74
6.2.2	Authentication	74
6.2.3	Core Functionalities	75

7 Conclusions	80
7.1 Obtained results and implications	80
7.2 Future improvements	81
7.3 Learned lessons	81
Bibliography	82
Acknowledgements	85

List of Figures

1.1	Execution process with Just-In-Time compiler	4
1.2	The project structure in a Xamarin solution compared to that of an MAUI solution; the former is composed of multiple projects.	5
1.3	The difference between MAUI and <code>Xamarin.Forms</code> approach in the rendering of cross-platform components	6
1.4	The application of the layered architecture architectural pattern in the MAUI framework	8
1.5	The MVC pattern scheme.	11
1.6	The MVVM pattern scheme.	12
1.7	The MVU pattern scheme.	16
1.8	The <code>MessagingCenter</code> class provides multicast publish-subscribe functionality	17
1.9	Different binding modes supported by the MAUI framework.	19
2.1	The Mole Vanvitelliana on which the museum is located today	22
2.2	The Omero Museum access ticket	23
2.3	The first two pages of the Omero Museum questionnaire	24
2.4	The activity diagram depicting the museum's internal activities	25
3.1	The use cases diagram of the staff application	30
3.2	The use cases diagram of the customer application	34
3.3	The mapping matrix	35
3.4	The analysis class diagram	37
4.1	The <i>Realtime Database DBMS</i> overall structure	40
4.2	The screens map for the customer application	41
4.3	The screens map for the staff application	42
4.4	The mockup for the staff application's home page	43
4.5	The mockup for the staff application's statistics page	43
4.6	The mockup for the staff application's ticket office page	44
4.7	The mockup for the staff application's artworks page	44
4.8	The mockup for the staff application's chats page	45
4.9	The mockup for the staff application's account page	45
4.10	The mockup for the customer application's login, home and chat pages	46
4.11	The mockup for the customer application's tickets, ticket office and statistics pages	46

4.12	The mockup for the customer application's account, exhibit and questionnaire pages	47
4.13	The design class diagram for the managers classes	48
4.14	The design class diagram for the model's classes	49
4.15	The design class diagram for the view and view model classes of the customer application	50
4.16	The design class diagram for the view and view model classes of the staff application	51
4.17	The sequence diagram for the <i>Login</i> use case	52
4.18	The sequence diagram for the <i>ResetPassword</i> use case	52
4.19	The sequence diagram for the <i>UploadAvatarImage</i> use case	53
4.20	The sequence diagram for the <i>SignUp</i> use case	53
4.21	The activity diagram for the <i>ExhibitsCUD</i> use case	54
4.22	The activity diagram for the <i>SendMessage</i> use case	55
4.23	The activity diagram for the <i>ValidateTicket</i> use case	55
5.1	Installation of MAUI framework in Visual Studio Installer 2022	57
5.2	The <code>App.xaml.cs</code> file representing the software entry point	58
5.3	The three <i>NuGet</i> packages that integrate the corresponding Firebase services	58
5.4	The <code>LoadJsonObject</code> method of the <code>DatabaseManager</code> class used to retrieve an instance from the remote database	59
5.5	The <code>LoadJsonArray</code> method of the <code>DatabaseManager</code> singleton class used to retrieve a list of instances under a specified node on the remote database	59
5.6	The rules managing the access to the <i>Realtime Database</i>	60
5.7	The creation of the <code>FirebaseClient</code> object with the user token for recognition	60
5.8	The <code>IconFont</code> class containing a list of all the Material symbols	61
5.9	The registration of used fonts	62
5.10	The different themes for the staff application	62
5.11	The implementation of the menu item in the staff application	64
5.12	The visual behaviour of the flyout menu item	64
5.13	The implementation of the Sharpnado's <code>ViewSwitcher</code> class	65
5.14	The implementation of the Sharpnado's <code>TabHostView</code> class	65
5.15	The <code>ScanQrCode</code> method implemented with the <i>ZXing</i> library	66
5.16	The popup delivered with the <i>Mopups</i> library	66
5.17	The custom implementation of the <code>IWindow</code> handler	68
6.1	Developer mode is required for the installation of sideload applications	69
6.2	The procedure to follow to install the custom certificate	70
6.3	The installation of the <i>msix</i> package for the staff application	70
6.4	The staff application login page	71
6.5	The log-out button on the top menu of the staff application	71
6.6	The statistics on the weekly ticket validation purchase ratio	72
6.7	The procedure for scanning a QR code	72
6.8	The artwork collection	73
6.9	The chat page	73
6.10	The theme selection buttons on the settings screen	74
6.11	Enabling app installations from unofficial sources on Android 8.0 and onwards	74
6.12	The customer application authentication screens	75
6.13	The logout button on the top bar of the customer application	75
6.14	The room selection buttons	76
6.15	The artworks and exhibits overview pages	76

6.16	The bought tickets list's filter	77
6.17	The ticket office purchase procedure	77
6.18	The customer's chat filter feature	78
6.19	The customer's personal statistics page	78
6.20	The questionnaire's compilation page	79
6.21	The customer's account page	79

List of Tables

1.1	Glossary of terms concerning the rendering of graphical visual elements. . . .	6
1.2	The four parameters required to establish a data binding link between a component attribute and a property of a binding context.	18
3.1	Functional requirements table for the internal management application	27
3.2	Functional requirements table for customers' application	28
3.3	Non-functional requirements table	29

In recent years, cross-platform frameworks have become increasingly significant in the realm of software development. Following on from the Xamarin experience, Microsoft plans to boost its market share with the launch of .NET Multi-platform App UI (MAUI). This framework provides a powerful solution for building cross-platform applications adopting a single codebase, catering to the growing need for apps that can operate seamlessly on different devices and operating systems, such as iOS, Android, Windows, macOS, and others. Essentially, it represents the result of Microsoft's efforts to pursue its Write Once, Run Anywhere (WORA) vision. This technique reduces development time and effort, making it a cost-effective option for businesses aiming to extend their reach across various platforms.

The abstraction from the execution environment of devices has led to an increase in job opportunities for .NET developers, rather than specialised device developers. Thus, efforts invested in learning .NET result in the benefit of a proficient tool capable of assembling various types of projects.

One of the key advantages of .NET MAUI is its native UI capabilities. It provides access to native controls and features, ensuring that applications look and perform as if they were developed specifically for each platform. This native-like experience provides better control over OS-specific functionalities, minimal execution overhead and preserves original visual and functional consistency.

The situation differs when considering web or hybrid approaches implemented by commonly used frameworks, including Electron, Ionic and Cordova. These methods depend on the interoperability of web browsers, resulting in less efficient and less adaptable applications. This indicates that the commonly employed languages are HTML, CSS, and JavaScript.

In addition to its technical capabilities, .NET MAUI benefits from a strongly growing developer community. This means that developers can find a wealth of resources, documentation, and third-party libraries to streamline their development process and overcome any challenges they may encounter.

In conclusion, as the demand for cross-platform applications continues to grow, .NET MAUI is positioned to play a central role in shaping the future of software development.

This thesis showcases the development of a museum management software, aimed at improving service quality. The software comprises two applications, one serving as an interface with the facility's assets management for staff employees, and the other meant for its customers, streamlining interaction with the services offered. Following this, a meticulous design phase will be proposed, based on the UP methodology, satisfying the gathered requirements. Furthermore, an investigation into the most distinctive features addressed during the implementation phase will be presented, accompanied by a user manual as supplemental

documentation for the project. Finally, it will be demonstrated how the development of this application serves as a modernisation attempt for the museum structure involved.

In detail, this thesis comprises seven chapters, structured as follows:

- Chapter 1 presents a comprehensive overview of the .NET MAUI technology, complete with theoretical aspects critical for its basic usage.
- Chapter 2 provides a brief contextualisation of the museum reality in which the software to be developed will operate.
- Chapter 3 examines the identified requirements and precisely outlines the use cases that need to be fulfilled.
- Chapter 4 presents a sturdy design based on the requirements defined previously. Database design, screen mockups, and refined class, sequence and activity diagrams will be presented.
- Chapter 5 outlines the implementation phase highlights of the two applications, with a focus on listing encountered issues related to the premature state of the framework.
- Chapter 6 provides detailed user manuals for both applications, including comprehensive installation and setup guides.
- Chapter 7 presents the dissertation's concluding remarks, reviewing the lessons learned from analysing the development phase errors and exploring potential future enhancements.

The MAUI cross-platform framework is a developing technology from Microsoft that has significantly improved with the release of .NET 7. Although it still contains some minor issues, MAUI provides a reliable foundation for developers to build cross-platform applications.

In this chapter, we first provide a brief overview of the underlying technology of the .NET world, leading up to the birth of the MAUI framework.

Then, we delve into more detail on the MAUI framework, providing a panoramic of its main features. Most of these features were first introduced during various Microsoft conferences in early 2020 when MAUI wasn't already released. Finally, we explore the three fundamental concepts every developer interested in creating a MAUI application should understand.

1.1 An Overview of the .NET World

Before diving into the MAUI framework, it's worth spending a few words about the .NET development platform and what factors contributed to the creation of MAUI. It is worth noting that Microsoft has always adhered to the "one .NET" vision. The goal is to provide a multi-platform framework with a single runtime for all devices and operating systems, such as Android, iOS, and Linux, and was finally achieved with the release of MAUI.

1.1.1 What is .NET?

As Microsoft's documentation states, .NET is a "free, open-source, cross-platform framework for building modern apps and powerful cloud services". The "cross-platform" aspect was the most difficult and took multiple steps over time, with a significant contribution from the "open-source" community. In the beginning, .NET was a Windows-centered non-open-source framework and the task of sharing code between different platforms i.e., the so-called Write Once, Run Anywhere (WORA), wasn't easy because of how it was built.

On the other hand, Sun Microsystems, with its first version of Java in 1996 dominated the cross-platform market, providing not only a programming language but also a virtual machine, i.e., the Java Virtual Machine (JVM). It completely fulfilled the WORA dream also thanks to the hard work of Java's team to provide a runtime version of the JVM for almost any existing operating system.

Microsoft also tried to join the cross-platform race. The paradigm used was the same: the .NET Framework exposed the .NET Base Class Library (BCL) as an abstraction layer between the developer's code and the Windows operating system.

With the release of .NET Core, an implementation of the .NET Standard, it became possible to run a .NET application on different operating systems. Linux, and so Android and iOS with their Unix-like kernels, were also supported by .NET Core thanks to the Mono runtime, which was developed by the open-source community.

1.1.2 Compilation and runtime process

In the .NET framework, each supported programming language has a dedicated compiler that converts developer code into MSIL code, which is the .NET framework's lowest-level programming language. (Figure 1.1).

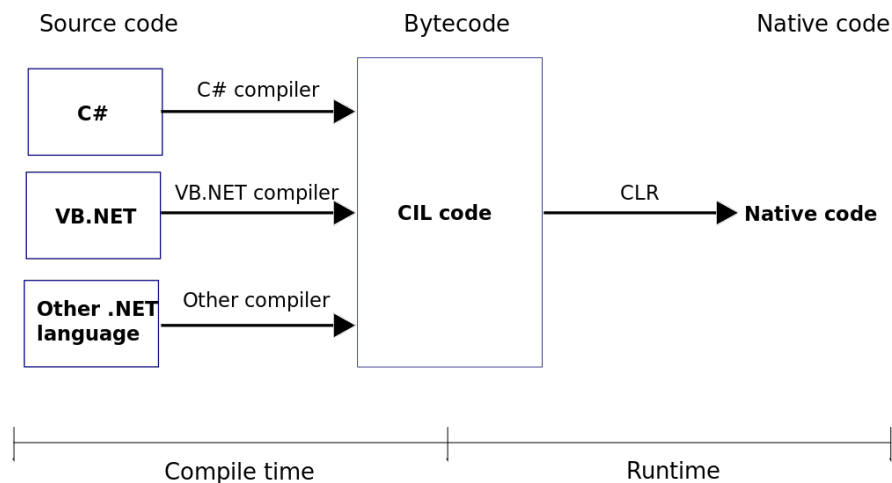


Figure 1.1: Execution process with Just-In-Time compiler

The execution process, on the other hand, involves two ways of converting MSIL code into native code. The first approach is to use a Just-In-Time (JIT) compiler, which is dynamic and occurs continually while the software is running. The second approach involves using CoreRT, the .NET Ahead-Of-Time (AOT) compiler, which is a one-time operation that must be completed only once to generate an executable file for the specified platform.

1.2 The birth of MAUI

The release of .NET 6 (long-term support), in November 2021, made Microsoft a real competitor in the cross-platform industry. .NET Multi-platform App UI (MAUI) is now included in the bundle, and any developer can now start writing multi-platform native, web, or hybrid apps on the spot. It can be said that MAUI was developed as an evolution of Xamarin.Forms which was already six years old.

1.2.1 Project structure

The single-project structure is certainly one big change from the Xamarin divided-project structure that MAUI took. Developers who used Xamarin had to deal with multiple projects, as shown in Figure 1.2, one for each platform they wished to support, which led to a complex project solution. This sure is easy to become messy in elaborate projects. Naturally, this does not imply that writing platform-specific code is no longer possible in MAUI; rather, it is still possible to do so inside the "Platforms" folder, which contains all the files required to make the application work on each device, such as the "AndroidManifest.xml" for Android OS.

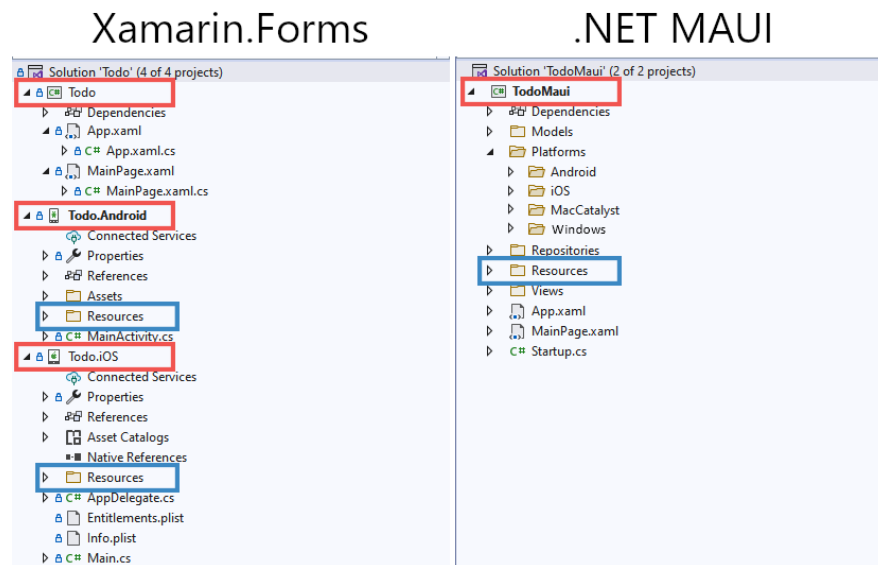


Figure 1.2: The project structure in a Xamarin solution compared to that of an MAUI solution; the former is composed of multiple projects.

Resource redundancy is another problem that MAUI was able to fully solve. In Xamarin, each platform-specific project had its own "Resources" folder, which means that every time an image needs to be imported, it must first be exported in different size formats, by device-specific rules. The same applies to font assets.

MAUI managed to fix all these tedious tasks by hiding the resources management from the developer; a single "Resources" folder can now be found. All the developer needs to do is importing the file (preferably a vector image) into the folder, and the framework will automatically take care of the resource resizing and adaptation for each specified platform.

1.2.2 WinUI and MacCatalyst

After the unifying process taken by Microsoft's team with .NET 5, desktop applications (Windows and MacOS platforms) were also supported by Xamarin, thanks to the interoperability of the two distinct BCLs. However, due to the insufficient support provided, it was unusual for a developer to choose Xamarin when developing a desktop-based application.

MAUI, on the other hand, come with WinUI 3 support, which is a UI framework built on top of Universal Windows Platform and includes both controls and aesthetics to improve the so-called "look and feel" of applications. It also includes Fluent Design, an open-source, cross-platform design system developed by Microsoft in 2017.

MAUI provides support for MacCatalyst too. MacCatalyst is designed to transfer iPad apps onto macOS, much like UWP did for Windows, while keeping the native nature of the application and utilizing all the macOS-specific capabilities.

1.2.3 A loosely-coupled approach

MAUI rebuilt from scratch some of the mechanisms of Xamarin. One of these is the rendering of controls. Table 1.1 shows the definition of the components involved in this technology.

Each control defined inside the `Xamarin.Forms` namespace needs to have a renderer for each supported platform. This implies that each renderer is tightly coupled with the corresponding control of the framework, which results in a structure that is not very flexible.

Name	Definition
Control	The cross-platform UI element. It is an abstraction that the framework implements to allow the developer to reuse that component across several platforms.
Handler	The component in charge of using a native platform control to implement a cross-platform request.
Mapper	A dictionary mapping the cross-platform framework API to a specific platform API.
Renderer	The Xamarin platform-specific implementation of control. In other words, it is the component that must be created whenever a new custom control is required. MAUI strongly advises developers against using it while retaining compatibility for a smoother transition from Xamarin.

Table 1.1: Glossary of terms concerning the rendering of graphical visual elements.

This strong bond between the two components was due to both the presence of Controls' references inside Renderers and the usage of `INotifyPropertyChanged` interface in order to communicate the occurrence of a change on a property. All of this results in slow Assembly scanning and in the difficulty in reaching the native platform.

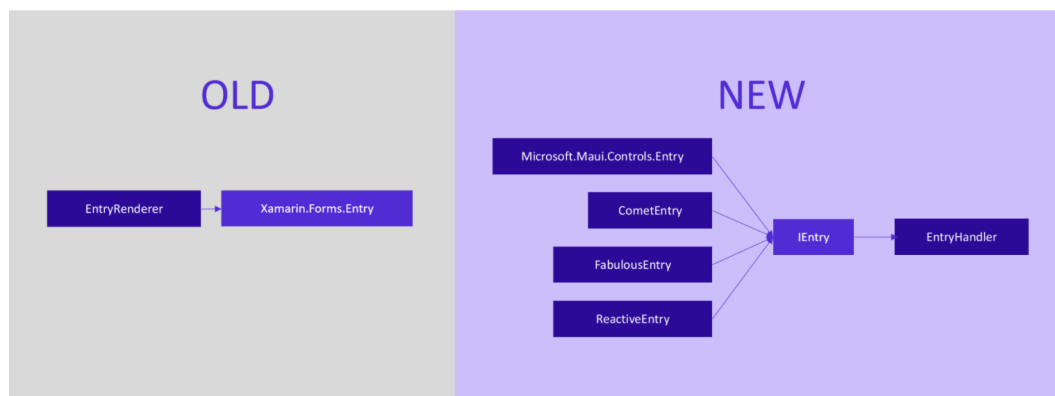


Figure 1.3: The difference between MAUI and `Xamarin.Forms` approach in the rendering of cross-platform components

As shown in Figure 1.3, MAUI's more recent structure exposes an interface that is implemented by various classes that represent various iterations of the same Entry control. This interface serves as a mapper between the control's properties and that of one of its platform-specific implementations and is referenced by a handler with which the framework will interact. If the developer ever needs to add a new implementation of the entry control, all he needs to do is adding a new implementation of that interface. All connections are explicitly declared to minimize assembly scanning runtime and reduce app startup time.

Based on software engineering theory, what MAUI did was nonetheless the application of the "Dependency Inversion" principle, which states that high-level classes should depend on low-level ones. This paradox is solved by the interposition of interfaces between the two layers, which are exposed by the high-level classes. This flexibility allows for the addition of new components to low-level classes by simply adding an implementation of the corresponding interface and referencing an instance of it in higher-level classes. In this way, the Open-Close principle is adhered to without the need to edit even a single line of high-level code.

1.2.4 Dependency injection

Dependency injection enables the decoupling of concrete types from the code that depends on them. Most often, it makes use of a container that contains a list of registrations and mappings between interfaces and abstract types, as well as the concrete types that implement or extend these types [Goldman, 2023].

The most basic example of a dependency injection is the creation of an object; it happens when a class constructor is invoked and some values needed by an object are passed as arguments. This injection is done by injecting the dependencies the object needs into the constructor, which does not know the class responsible for instantiating those objects or their runtime concrete types.

To make use of MAUI's dependency injection feature on a class, it must first be registered in the dependency injection container. This can be achieved by specifying the interface or abstract type along with its corresponding concrete implementation. The class can then be registered as a Singleton or Transient, depending on whether the dependency injection system should store a reference of the instance.

Dependency injection is an implementation of the "inversion of control" pattern, which asserts that the management of an application's flow or lifetime should be transferred to a different part of the application to benefit in terms of flexibility. It is worth pointing out that the support for dependency injection, which is provided for both the MVVM and MVU patterns (see section 1.4), is not intended to compel developers to use this approach every time they build a view or view model class. Rather, the code should follow the dependency injection approach only when the class of interest has any dependencies or is a dependency for other classes. The abuse of dependency injection only leads to unnecessary project complexity.

1.2.5 Hot reload

Hot reload is a highly useful .NET technology that Microsoft created to increase efficiency. It significantly reduces the frequency of program restarts during the development phase. In case of minor modifications, only one click is required to refresh the running application, bypassing the time-consuming compilation procedure. In case of radical changes to the code, like the creation of a class or a method, the MSIL bytecode needs to be rebuilt and thus the application must be recompiled.

Xamarin began to experimentally implement XAML hot reload capability in its recent releases, whereas .NET hot reload was not supported at all. MAUI, on the other hand, included full XAML and .NET compatibility.

1.2.6 MAUI's compilation process

MAUI uses a different compilation strategy depending on the platform on which the application runs, in any case, a native executable file is delivered. In detail the adopted strategies are the following:

- When creating an Android application, C# and XAML code are first turned into MSIL bytecode, which is then just-in-time compiled into a native assembly. As mentioned in Section 1.1.2, this can result in a delayed startup time, but since MSIL is a managed code, better optimization can be accomplished.
- iOS apps, on the other hand, are fully ahead-of-time compiled into native ARM assembly code. This can result in a faster startup; however, since ARM is unmanaged code, no further optimization can be accomplished during runtime; nevertheless, compile-time security checks are always available.

- MacOS apps are built using the MacCatalyst technology, which, as shown in Section 1.2.2, is able to port iOS apps directly to the Mac desktop world. In other words, it combines the Mac APIs from AppKit with the iOS app development framework UIKit.
- Windows apps are built using the WinUI 3 framework, which represents the evolution of UWP.

Of course, this complex compilation and execution procedure is fully hidden to the developer.

1.2.7 MAUI's layered architecture

In conclusion, it is worth mentioning the architecture adopted by the MAUI framework. It comes as no surprise that the overall structure is very similar to that of Xamarin, particularly for the core at its bottom.

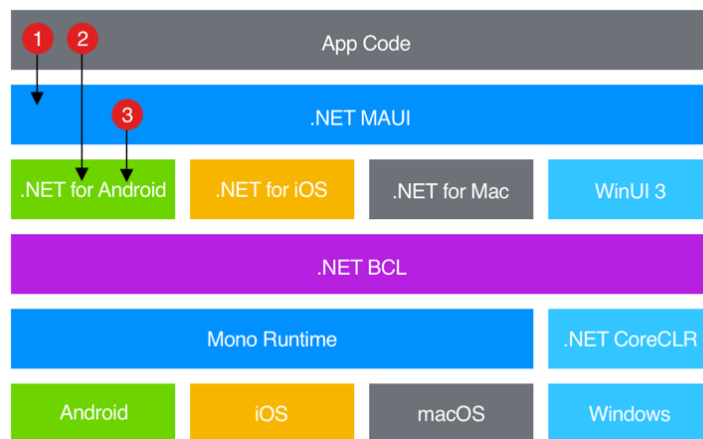


Figure 1.4: The application of the layered architecture architectural pattern in the MAUI framework

As shown in Figure 1.4, .NET MAUI is a high-level abstraction layer with which the developer should interact the most. Each call to MAUI's API (1) is automatically handled by the framework and is completely cross-platform. In turn, MAUI will be responsible for consuming the native platform APIs (2). However, it is not forbidden for developers to directly access platform-specific APIs to satisfy special needs where no support is already provided by the framework (3).

In short, the advantages of this architecture lie in the "separation of concerns" realized with the stratification of components. This is one of the basic principles of software engineering, arising from the algorithm design paradigm "divide et impera", aimed at the minimization of interdependencies between different components of the software.

1.3 XAML

"The eXtensible Application Markup Language (XAML) is an XML-based language that's an alternative to programming code for instantiating and initializing objects and organizing those objects in parent-child hierarchies. XAML allows developers to define user interfaces in .NET Multi-platform App UI (.NET MAUI) apps using markup rather than code" [Britch, 2022c].

1.3.1 The limits of XAML

XAML is a declarative markup language which simplifies the creation of UI pages for a .NET MAUI application. According to its definition, the usage of XAML is not mandatory. This implies that anything we include in our XAML code can also be implemented in the corresponding "code-behind" C# file linked to each XAML file. Essentially, XAML serves as an alternative approach to creating instances of non-abstract classes with a public default constructor and setting their properties. Therefore, it is important to weigh the benefits of using XAML.

First of all, XAML is often more succinct and readable than equivalent code [Britch, 2022c]. The readability is provided by the inherited XML features, which mirror the parent-child structure of UI components with improved visual clarity. Furthermore, XAML is well suited to work with the MVVM pattern, making it relatively simple to achieve the loosely coupled communication between a view and its view model (see Sections 1.4.1 and 1.5).

Another great XAML feature is its integration with C# 10's source generators, which is the ability to inspect user code as it is being compiled and to create new C# source files on the fly [BillWagner, 2022]. When a tag with a defined name (the `x:Name` property) is declared in a `.xaml` file, the compiler stores its reference in a new variable inside the generated class file and initializes its value in the `InitializeComponent` method, which is usually invoked in the code-behind class' constructor. In this way, we can access the in-view defined element from our code behind as we like.

It is crucial to remember that, in order to keep two classes with the same name, both of them must be marked as `partial`. That is the reason why the code-behind classes are so structured when UI is built with XAML.

However, we cannot completely replace our C# code with XAML. That is because there are several limitations to its usage. Things like logic, event handlers, loops, and conditional processing cannot be achieved in XAML. Thanks to the data binding feature supported by XAML and its markup extensions, simple logic can still be implemented directly in our XAML UI code. To summarize, XAML is a very good strategy for separating static UI declarations from code and event processing, which is always a good approach, as the separation of concerns principle indicates.

1.3.2 XAML markup extensions and shared resources

Compared to XML, XAML has some unique syntax features which are not defined in XML. The most important are:

- *Property elements* is the capability to specify a tag property's value with an object rather than a simple string, which will be implicitly converted to an instance of the required class. This is not possible in XML where the value of an attribute needs to be a string, with the only possible variation being which string encoding format is being used.
- An *attached property* is intended to be used as a type of global property that is settable on any dependency object [George, 2022]. The attached properties are useful to enable child elements in a markup structure to report information to a parent element without requiring an extensively shared object model across all elements [Andy De George, 2022]. This feature is especially useful when dealing with node¹ layouts.
- *Markup extensions* enable properties to be set to objects or values that are referenced indirectly from other sources [Britch, 2022d]. However, data bindings are where XAML markup extensions are most useful.

¹A node is composed of a tag, its attributes and all its inner content, including all of its child elements.

When writing XAML code, it often happens that we have to reference values defined somewhere else or which might require a little processing by code at runtime. "Hard-coded" strings aren't of any help in these kinds of situations. Rather, markup extensions are what we can rely on. In order to use a markup extension on an attribute's value we can use curly braces, { and }, and specify the needed resource. The resources, instead, can be defined either inside a `Resource` tag, which is common to all MAUI controls, since they all inherit that property from the `VisualElement` class, or inside a C# static class or enumeration. In both cases, these values can be accessed using markup extensions.

For instance, we can use markup extensions to share a common style across multiple controls, rather than copy-paste several properties on each control, causing excessive redundancy. This solution has mainly two benefits, namely:

- a more legibility of the code,
- the fact that properties are centralized, and thus it is possible to one-shot edit all controls adopting the style.

It is important to note, that the scope of a `Resource` element is limited to the children of the tag in which it is defined. If, instead, we need to give public visibility to a resource, we can do so by writing it inside the `App.xaml` file, which is the parent of all user-defined views.

Each item in a resource dictionary needs to define a `x:Key` attribute. In this way, it can be referenced everywhere by a specific view in its scope. In order to access a resource inside an element's property, we can use the markup extensions `StaticResource` and `DynamicResource` based on whether the resource may or may not change at runtime.

1.3.3 C# Markup

C# Markup is a set of fluent helper methods and classes designed to simplify the process of building declarative .NET Multi-platform App UI (.NET MAUI) user interfaces in code [Shaun Lawrence, 2022].

As we are aware, it is possible to write UI views directly in our C# code-behind. However, if multiple properties are initialized after a control is declared, the syntax may become verbose. To address this issue, the C# Markup extension is included in the .NET MAUI Community Toolkit Markup package. To use the C# Markup extension, one must install the corresponding *NuGet* package called `CommunityToolkit.Maui.Markup`. This extension improves the syntax's fluency by enabling method chaining and providing a variety of tools to simplify common tasks. By leveraging the convenient *C# extension functions* feature, the toolkit extends the methods of its components and modifies them to return the instance rather than `void`. As a result, it becomes possible to call another method immediately after the previous one, resulting in a more streamlined syntax. In Section 1.4.3, we will see how this feature is essential when dealing with MAUI applications that follow the MVU architecture.

1.4 Architectural patterns

As software engineering teaches, patterns are consolidated solutions to recurrent problems. In particular, architectural patterns refer to strategies that address the structure of an entire project, rather than just some specific classes within it. Novice programmers may be inclined to adopt a monolithic architecture, which involves placing most of the code in a single C# file. Although this approach may seem like the quickest solution, it is a highly rigid and inflexible one that is only suitable for small use cases. When we encounter a bug in our

application, we must determine where the problem resides within our codebase and identify the lines of code which are responsible. This task becomes increasingly challenging as the project grows in size. Furthermore adding new features to our application runs the risk of damaging our previously written code.

Modern development frameworks encourage developers to adopt established patterns in their code, offering numerous benefits despite the need for an initial effort. By clearly dividing responsibilities, these architectures are easier to comprehend than the monolithic design, saving time when implementing changes. Moreover, adopting these patterns correctly ensures the security of the codebase when adding new features. As the "Open-Closed" principle says, our classes should be open for extension but closed for modification [2]. Hence no line of already written code should be edited when a new feature is added.

There are many established architectural patterns, such as the Client-Server, Layered Architecture, Pipe and Filter, and Model-View-Controller (MVC), each serving a specific purpose. We previously explored the use of the Layered Architecture pattern in the design of the MAUI framework. However, when developing applications, a different approach is often taken. Applications can be viewed as user-friendly tools that display and modify a centralized data source, such as a database management system or a backend with filesystem storage, which may be accessed by various software targeting different platforms. When multiple clients access a shared resource, changes made by one client should be visible to all the other clients that request the same resource. The MVC pattern is perfectly suited to meet these needs.

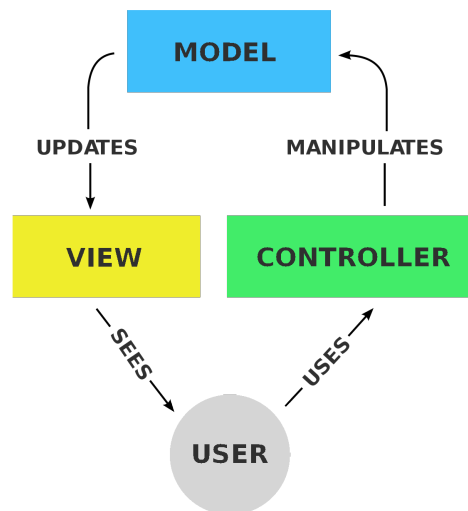


Figure 1.5: The MVC pattern scheme.

The MVC pattern, whose schema is illustrated in Figure 1.5, separates an application's logic into three interconnected components. The model represents the data and the business logic. The view displays the data to the user in a visually pleasing way. The controller handles user inputs and updates the model and view accordingly. The model can be considered as the fixed part; it is shared between different applications. The Observer design pattern serves as the unique means of communication between the view and the model; a view subscribes itself to a model's list in order to be notified of any data changes.

Although MVC offers a good strategy for designing an application, it also presents some weaknesses. Over time, several MVC variants were developed; some of these are the Model-View-ViewModel (MVVM) and the Model-View-Update (MVU) patterns. They both are very similar to their father, the MVC pattern, but at the same time, they provide a more

flexible and maintainable architecture, with better support for data binding and reactive programming (the hot-reload MAUI’s feature, for instance). That is why MAUI framework provides support to both those patterns.

1.4.1 MVVM

Today’s modern frameworks, such as Android Studio, Angular and MAUI, adopt the MVVM pattern as their preferred architecture. The MVVM pattern, which structure can be seen in Figure 1.6, was introduced in 2005 by Microsoft architects Ken Cooper and Ted Peters [Vermeir, 2022] and has become popular because it offers several advantages over the older MVC pattern. One of these is that it provides a better separation of concerns between the different components of the application. In MVC, the controller is responsible for both handling user input and updating the view, which can lead to tight coupling and difficult-to-maintain code. In contrast, MVVM separates the responsibilities of the view and the view model, with the view model serving as an intermediary between the view and the model. More specifically, the view “knows about” the view model, and the view model “knows about” the model, but the model is unaware of the view model, and the view model is unaware of the view. As a result, the view model separates the view from the model, allowing the model to change independently of the view. This leads to a more modular architecture.

The view model serves a similar purpose to that of the controller in the MVC pattern, but with a notable distinction: it is a stateful component. This is because it holds data and values which may change during runtime.

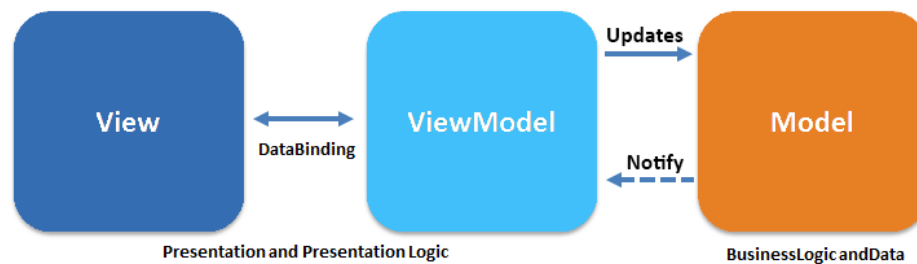


Figure 1.6: The MVVM pattern scheme.

The MVVM pattern enables designers and developers to work asynchronously. According to this pattern, the view should not contain any business logic. Hence, the designer’s responsibility is only to provide a UI to display data, without worrying about how that data is obtained. Meanwhile, the developer can focus solely on implementing the business logic in the view model. The developer ensures that the data obtained from the model is correctly formatted according to the designer’s requirements. In addition, when working with XAML, designers can build the app UI without needing to modify much of the C# code. However, in certain situations, the code-behind might need to contain UI logic that implements visual behaviours challenging to express in XAML, such as animations.

The interaction between views and view models is mediated by data binding (see Section 1.5). This is another advantage of MVVM; data binding can simplify the process of updating the user interface. There are various techniques available to create and connect views and view models. They can be broadly categorized into two types: *view-first composition* and *view-model-first composition*. The choice between these two techniques usually comes down to personal preference. However, the ultimate goal of all these techniques is to ensure that the view is associated with a view model. This means that the `BindingContext` views’ property must be initialized with an instance of a view model class. The choice between these

two techniques determines how our project's architecture is conceptually intended.

- A view-first composition is an approach in which the app is composed of views that connect to the view models they depend on. This means that the view models have no direct dependence on the views themselves, resulting in a loosely coupled structure. As a result, the view models become interchangeable parts that are used by the views, which are the real subjects of the application.
- With view-model-first composition, the app is built around the view models, with a service that is responsible for finding and connecting the appropriate views to the view models. This approach may feel more intuitive for some developers, as the creation of views can be abstracted away, enabling them to focus on the app's logical structure without worrying about UI specifics. It also allows for the creation of view models by other view models. However, this approach can be more complex, making it challenging to understand how the different parts of the app are created and connected.

Whenever we choose to adopt a view-first or a view-model-first composition, the MVVM components' overall content remains the same. Inside a view model, the properties that need to be accessed by the view must have public visibility, as the data binding system needs to access their values from outer classes. A common practice is to create a private variable with the desired value and a public property that mediates access to that variable. In this way, any changes made to the property are properly propagated to all observers (the controls which bind that property), ensuring that the view is updated accordingly.

It is important to note, though, that the data binding system does not constantly monitor its `BindingContext` for changes; this would simply cost too many resources. Rather, it trusts the developer to tell it when a value changes so that it can refresh that property; this is done by letting the view model class implement the `INotifyPropertyChanged` interface.

The `PropertyChanged` event is the only member exposed by that interface. That event takes the name of the property that just changed its value as a string parameter. Now, there are mainly three ways to specify it. In particular:

- We can pass the name as a "hardcoded" string. This is the quickest method, but it can make the code difficult to maintain in the long run, especially if the property name needs to be changed.
- We can use `nameof`, an useful C# operator, to retrieve the name of a property, instead of hardcoding its name as a string, making the code more maintainable. If we change the property name through a refactor, Visual Studio will automatically update the name inside the `nameof` operator, which can save time and prevent errors. It is worth noting that runtime performance is not impacted in the slightest when using this approach, because the conversion from the property to its name as a string occurs during the compilation time.
- Lastly, the most elegant way is to use the `CallerMemberName` attribute on the parameter of the `OnPropertyChanged` method. By doing so, the compiler will automatically retrieve the name of the calling property and pass it to the method, eliminating the need to hardcode or use `nameof`.

1.4.2 MVVM Toolkit

The Microsoft Community Toolkit is a collection of *NuGet* packages offered by Microsoft that contains a plethora of useful features for MAUI developers. One of the toolkit's key

components is a library designed to simplify the implementation of the MVVM pattern for developers. As we saw in the previous example, implementing MVVM can be tedious, requiring the developer to implement `INotifyPropertyChanged` everywhere, among other things. The MVVM Toolkit takes advantage of the C#'s source generators feature to automatically generate a large portion of the code and implement binding using a caching system, resulting in optimal performance. The toolkit inspects our view model code and generates classes specifically designed to handle and notify property changes. As explained in Section 1.3, this implies that our view model class cannot exist on its own and must be marked as partial.

Practically, in order to use the toolkit inside our view model class, we need to make the view model class extend the abstract class `ObservableObject`, which in turn implements the `INotifyPropertyChanged` interface we need.

The code of a view model class that utilizes the MVVM toolkit appears much neater and more elegant. The primary distinction is that, for each property, the view model should have, we no longer need to implement the entire field structure, but only the private part. However, a question may arise: Is it no longer possible to incorporate any logic before or after changing the value of the variable? Prior to the MVVM toolkit, we were required to specify the setter method, which allowed us to add any desired logic to it. Actually, the toolkit's architecture is completely modular. This implies that we can override or implement partial methods on the fly. So, in order to answer the previous question, there are two different strategies to inject our custom business logic into the setter. Specifically:

- We can override the `OnPropertyChanging/OnPropertyChanged`, which are both inherited from the `ObservableObject` superclass, in our view model class. This implies, however, that these methods are common to all properties in the class; so we need to use a switch statement to distinguish between the changing/changed property name.
- A more elegant solution is offered by the MVVM Toolkit since its third preview of Version 8. When we use `[ObservableProperty]` to generate observable properties, the toolkit will automatically create two partial methods without any implementations [Pedri, 2022]. These methods are `On<PROPERTY_NAME>OnChanging` and `On<PROPERTY_NAME>OnChanged`. Thus, it is possible, in our view model code, to write a partial method with the same signature and write our custom logic onto it.

The MVVM Community Toolkit offers a lot of other helpful tools for developers. For instance, the `RelayCommand` class implements the `ICommand` interface and is utilized when we want to bind a method with a control's "Command" property in the view. On the other hand, the `ObservableCollection<T>` is a subclass of `Collection<T>` that triggers a `CollectionChanged` event whenever an item is added to or removed from the list. UI components, such as a `CollectionView` in MAUI, observe this event and refresh their list accordingly.

1.4.3 MVU

The Model-View-Update (MVU) architecture, also known as The Elm Architecture based on where it first gained popularity, is a simple and intuitive design pattern that is well-suited for use with frameworks like Flutter, as well as in MAUI. In this architecture, the Model represents the application state, which is passed to the View for rendering. When the user interacts with the View, update methods that modify the Model are called. In turn, these models trigger updates in the View [Adam, 2019].

While its name may suggest some similarities with the MVVM and MVC patterns, it must be noted that the MVU pattern has almost nothing in common with them. MVU is an entirely different philosophy compared to MVVM, which is quite similar to the MVC pattern. This philosophy is based on the concept of immutability [Jean-Marie, 2022]. In the MVU architecture, the only component allowed to cause side effects, i.e. if it modifies some state variable value(s) outside its local environment [5], is the so-called "command". All other parts of the program must be free of side effects. This approach not only makes most of the program very flexible but also makes it easy to reason about. Furthermore, the debugging process is more or less limited to exploring the code of the commands and is thus a simpler task.

It is worth noting that the MVU pattern originated in Elm, a programming language that follows functional programming principles. Functional programming is a computer science theory based on Lambda calculus, proposed by the mathematician Alonzo Church [3]. This theory states that every element in a program can be represented by a mathematical function, and programs are built by applying and composing these functions [4]. This implies that XAML is not well-suited to work alongside an MVU architecture. On the other hand, C# markup is highly compatible due to its method chaining feature, which we discussed in Section 1.3.3. Therefore, choosing between the two patterns for a project's structure is a crucial decision and cannot be easily switched once the project is launched.

In order to take a grasp of the MVU underlying idea, we first need to understand how each part is defined.

- The Model is conceptually different from the MVVM's Model. Rather than a class holding an arbitrary set of properties and methods, the MVU's Model represents the state of the application. This confusion has led a part of the community to refer to the model with the name "state".
- The View is responsible to describe the UI. To be more precise, the View represents the current state of the Model. Due to its loose coupling from the other components of the application, it is easy to test and is well-suited for MAUI's Hot Reload feature, which allows for the runtime deployment of view changes.
- The Update function is the only place that directly manipulates the Model. Differently from other approaches, it never mutates the Model, but instead creates an updated copy [Bandt, 2020]. When a user initiates a command on the UI, it triggers an update function. The update function executes the command, which may modify the Model's state. Since the Model is immutable, the update function creates a new modified copy of the Model and returns it. Subsequently, the view updates itself with the new instance of the Model.

The communication between the view and the update components is mediated by a message in order to achieve a loosely coupled architecture. This decision is based on the Command design pattern, which states that communication between components should be mediated through an object, the message, in order to minimize concerns [Shvets, 2019]. The message encapsulates the information that needs to be transmitted, which allows for the parameterization of methods with various requests. This approach provides a transformation that enables more flexible and dynamic communication between components.

In conclusion, it can be said that MVU is a completely functional way to build UI, and features a nice way to sharply separate the UI from logic. The overall architecture is resumed in Figure 1.7.

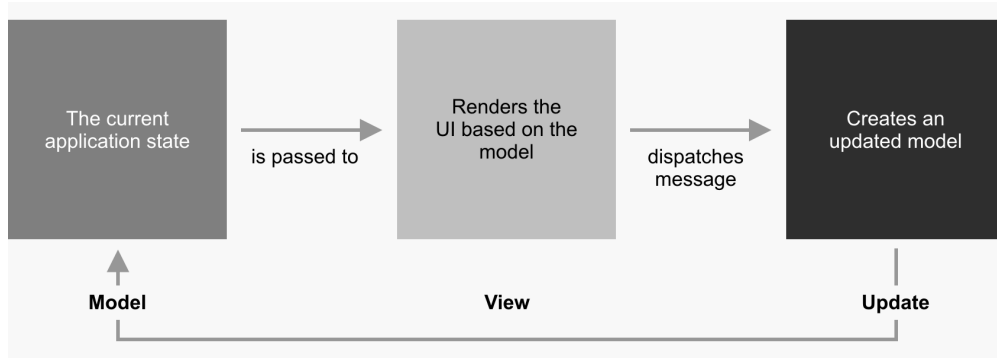


Figure 1.7: The MVU pattern scheme.

1.5 Data binding

Data bindings enable the linking of two objects' properties such that changing one affects changing the other. Although data bindings can be completely defined in code, as explained in Section 1.3.3, XAML offers convenience and shortcuts, making it a very valuable tool [Britch, 2023d]. Data binding goes very well with the MVVM pattern, where the view and the view model are often connected through data bindings.

1.5.1 Communication between loosely coupled components

Abstracting the concept, data binding is just one possible solution that the MAUI framework provides to solve the problem of communication between loosely coupled components. In software engineering, loosely coupled components are software components that are independent of one another and interact with each other through well-defined interfaces. However, communication between loosely coupled components can sometimes be challenging since they do not directly access each other's state or methods. For example, the first solution one may come up with is to keep a reference to the first class within the second one. Unfortunately, even though aggregation is a weaker bond than composition and hierarchy, it is still too strong. This approach may particularly hinder or postpone the garbage collection of short-lived objects of the first class. The reason is that its reference in the second class keeps it alive.

MAUI provides several solutions to facilitate information exchange between loosely coupled components, such as a view and a view model, or two view model classes in the MVVM architecture. The most significant solutions are the following:

- *MessagingCenter*, which has been deprecated in .NET 7 and replaced with *WeakReferenceMessenger* in the *CommunityToolkit.Mvvm* [Britch, 2022b], is an implementation of the publish-subscribe pattern. The latter enables publishers (the classes sending messages) to send messages without knowing their subscribers (the classes receiving messages). On the other hand, subscribers can listen to specific messages without knowing the publishers. This mechanism allows for communication between publishers and subscribers without requiring them to have direct references to each other. An overview of how message exchange occurs is provided in Figure 1.8. To achieve this outcome, the *MessagingCenter* class utilizes weak references. This implies that it does not maintain the objects alive and enables them to be removed by the garbage collector. Therefore, it should only be necessary to unsubscribe from a message when a class no longer needs to receive the message.

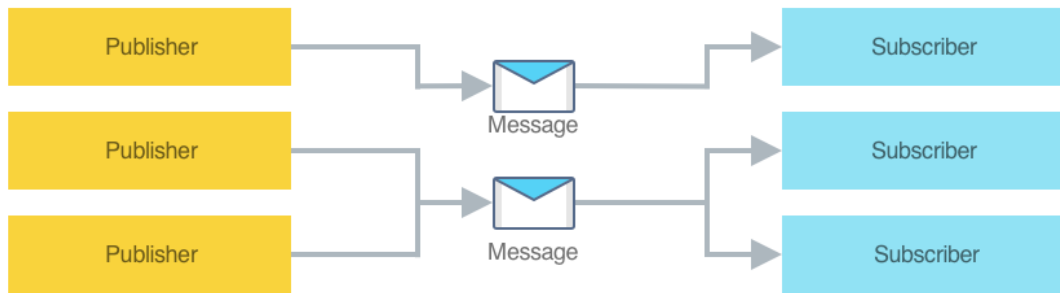


Figure 1.8: The `MessagingCenter` class provides multicast publish-subscribe functionality

- The MAUI *navigation system* facilitates communication between two view model classes. When coupled with the MVVM pattern, the common task of navigating between two pages can be challenging. In fact, it is hard to identify the destination view and its view model, while avoiding the creation of dependencies between the two views and so breaking the loosely-coupled MVVM architecture. Whether using view-first composition or view-model-first composition (see Section 1.4.1), the problem remains, as it requires specifying the name of a view or view model class to be instantiated from another view or view model class. Additionally, based on the context, the developer may need to trigger navigation from either a view or a view model, depending on the logic, so the navigation responsibility cannot be confined to one of the MVVM components.

The MAUI navigation system offers a solution to these issues by exposing the `INavigationService` interface and by providing services that abstract the process. The most popular is the `MauiNavigationService`, which is used to perform view-model-first navigation and is used in Shell-based apps, where `Shell` is the parent of all pages and stores the lists of all possible app routes. The good part is that, when triggering the navigation, we can also specify some arguments to be passed to the called view model.

- As mentioned in Section 1.4.1, the *data binding system* facilitates the exchange of information between a view and its corresponding view model, without tightly coupling them. To establish the link between the two, the `BindingContext` property must be set to an instance of a view model class (seldom, it can also be set to an instance of a model class). Every control in MAUI, including the `ContentView` and `ContentPage`, which serve as containers for a page or item template, inherits from the `BindableObject` class. The `Label` class, for instance, follows the following hierarchy path:

```
Label -> View -> VisualElement -> NavigableElement -> Element -> BindableObject
```

The `BindingContext` property of the `BindableObject` class allows for view model properties to be bound throughout the XAML file or the code behind. However, it's essential to understand that the data binding system is not just a means of storing a view model reference inside the view through the `BindingContext` property, as this would create a tight coupling between the two classes. Instead, the view just sets its data context through the `BindingContext` property and relies on the data binding system to update it automatically in response to changes in the view model. This way, the view is only aware of the properties that need binding, and the view model remains unaware of the view, resulting in two loosely coupled components.

1.5.2 The structure of data binding

As previously mentioned, to establish a link between a visual component (the target) and an object (the source), we need to specify the binding context. Since a visual component's `BindingContext` property is recursively inherited from its parent, it's a good practice to set it on the `ContentPage` or the `ContentView`, depending on the context. In most cases, the `BindingContext` is an instance of a view model class, aligning with the MVVM philosophy. However, when dealing with an item template, the `BindingContext` can also contain an instance of a model class, as the single item's instance represents the information contained in a model instance.

There are mainly two ways to assign the binding context in both cases:

1. One way is to assign the instance programmatically from the code-behind, usually done in the view's constructor. It is a best practice to do so after invoking the `InitializeComponent` method, which, as shown in Section 1.3.1, is responsible for parsing the XAML file and assigning references of non-anonymous components to variables. This approach can increase runtime performance because the binding system can immediately find components that need to establish a binding [6].
2. The second way is to assign the instance declaratively from the XAML code. This declarative construction and assignment of the view model by the view has the advantage that it's simple but has the disadvantage that it requires a default (parameter-less) constructor in the view model.

Name	Description
Target	The UI element involved which has to be a child of <code>BindableObject</code> .
Target property	The property of the target object. It must be of type <code>BindableProperty</code> .
Source	This is the source object referenced by data binding. It is inherited by the <code>ContentPage</code> 's or <code>ContentView</code> 's <code>BindingContext</code> property if it is not later overridden.
Path	The path to the value in the source object.

Table 1.2: The four parameters required to establish a data binding link between a component attribute and a property of a binding context.

After setting the binding context on a view, we can use the data binding on specific control's attribute using binding expressions in the XAML, which are part of the XAML markup extensions (see Section 1.3.2) or equivalently using the `SetBinding` method in the code behind. However, in both cases, there are some parameters that need to be set. These parameters are summarized in Table 1.2.

It's worth noting that the `Source` property is typically not required. In fact, it's only necessary if we want to override the inherited `BindingContext` and pick a context from a higher-level component in the view hierarchy.

In conclusion, it is a good practice to specify the `x:DataType` property on the top element inside our XAML code. It is simply a signal to the compiler that indicates the type of the `BindingContext` object that the view is using. While not strictly required, it does improve the IntelliSense experience by informing the compiler of the type at compile time rather than waiting until runtime to determine it.

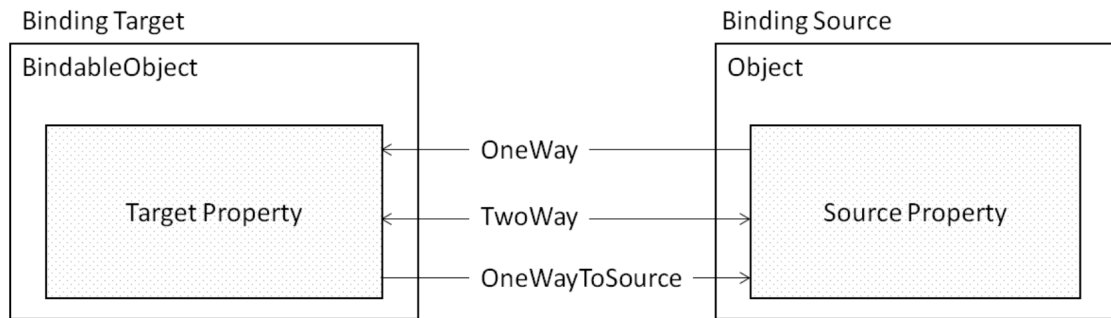


Figure 1.9: Different binding modes supported by the MAUI framework.

1.5.3 Binding mode

The data binding system is not restricted to displaying source object property on the target attribute; it can also work the other way around. When the target is an editable component such as an `Entry` or a `Picker`, it is possible to have the changes made by the user in the UI reflect on the source property. This is known as two-way binding. MAUI supports four different binding modes; three of them are represented in Figure 1.9. While each bindable property of a component has its own default binding mode, it can be overwritten in a binding expression by setting the "Mode" attribute to one of the following values:

- *OneWay*: this is usually used when statically presenting some data to the user. The communication is unidirectional, when its value changes, the source updates the target property.
- *TwoWay*: this is a binding mode that allows changes made to either the source property or the target property to automatically update the other. This type of binding enables bidirectional communication between the two properties. In other words, when the user edits the component, the new value of the target property is sent to the source property. Similarly, if the value of the source property changes, the target is notified, just like in the *OneWay* binding mode.
- *OneWayToSource*: this is the opposite of the *OneWay* binding mode, meaning that changes to the target property will update the source property. This mode is suitable when the source property is expected to be edited only by the user, and not by any other external entity. In such cases, using *OneWayToSource* instead of *TwoWay* binding can improve performance.
- *OneTime*: this is a variant of the *OneWay* binding mode that provides better performance by initializing the target properties from the source properties only once. Unlike the *OneWay* binding mode, any subsequent changes to the source properties won't be propagated to the target properties. This mode is not shown in Figure 1.9.

1.5.4 Binding value converters

When using data binding, it is common to have cases where the target and source property types differ. In such cases, transferring information between them is not straightforward, unless one type can be implicitly cast to the other. In MAUI, there are two approaches to solving this problem.

The first solution is to use the `StringFormat` property of a binding expression, which allows for specifying a string representation of the bound object by adopting standard rules.

For example, a floating-point number can be displayed with a specified number of digits after the decimal point, or a date can be shown as a string with a custom format.

For other types of conversions, the second solution comes in handy. It is possible to define a class that implements the `IValueConverter` interface, which enables custom conversion between source and target property types. Such classes are called value converters, but they are also often referred to as binding converters or binding value converters [Britch, 2023b]. When defining a value converter together with the `StringFormat` property inside a binding expression, the value converter is invoked before the result is formatted as a string.

The `IValueConverter` interface offers two methods for data conversion. The `Convert` method casts the source object, which is contained in the `value` parameter, to the `targetType` type, which is the type of the target property. The `ConvertBack` method performs the opposite operation. For `OneWay` and `OneTime` bindings, there is no need to specify any logic, and this method can simply return the `null` value. However, this approach is not recommended because value converters are often shared between different controls on the same page. Therefore, it is best to write the converter's code independently of the single binding expression in which it is used. It is a good habit to make the converters as generalized as possible too, maybe using generics instead of specific types.

1.5.5 Multi-binding

A group of binding objects can be connected to a single binding target property using multi-bindings. They are built using the `MultiBinding` class, which evaluates all of its `Binding` objects and returns a single value. Whenever any of the bound data changes, `MultiBinding` reevaluates all of its `Binding` objects.

Unlike single bindings (see Section 1.5.4), multi-binding objects require either the `StringFormat` and/or the `Converter` properties to be initialized. This is because the logic must be provided for selecting a single output value to assign to the target property. As stated, multi-bindings can also implement a converter. However, in this case, the `IMultiValueConverter` interface must be implemented instead of the `IValueConverter`

1.6 Wrapping up

The .NET MAUI technology is an incredibly valuable tool in the world of .NET. Microsoft's "one .NET" vision has been a great undertaking, but it has made learning the basics of MAUI relatively easy for developers who are already familiar with the .NET ecosystem. In the previous sections, we have provided with an overview of the main features of the MAUI framework, many of which are directly inherited from `Xamarin.Forms`. It is important for developers to have a solid understanding of these features before starting their first MAUI project.

The Omero Tactile Museum

The Omero Tactile Museum in Ancona is a unique institution that showcases art for the visually impaired. It is a pioneer in the field of tactile art and a model of accessibility and inclusion for other cultural institutions.

Within this chapter, we start by offering a concise historical overview of the museum's origins and the idea behind its establishment.

Subsequently, we delve into a more comprehensive exploration of the client visit experience and the underlying reasons for the need for digitalization in the museum's services. This phase serves to gain a better understanding of the problem domain and establish a set of requirements for the upcoming management software.

Concluding this chapter, we provide an analysis of the internal activities to ascertain the prerequisites that a management application must fulfil in order to respect the museum's business logic.

2.1 Foundation

The Omero State Tactile Museum in Ancona is a distinguished institution among the limited number of tactile museums worldwide. It offers a unique approach to experiencing art through touch, allowing visitors to perceive artworks with their hands. Founded with the aim of fostering inclusivity for individuals with visual impairments, the museum serves as an accessible space that welcomes all visitors.

Under the agreement signed on August 3, 2001, the museum is managed by the City of Ancona in collaboration with the Ministry of Culture.

2.1.1 The idea for an accessible museum

The museum was founded by Aldo Grassini and his wife Daniela Bottegoni, a blind couple with a passion for art who wanted to create a space where people could touch and appreciate art without barriers. In one of his interviews, Grassini stated:

"My wife and I, as avid travellers, have often had to contend with the absurd prohibitions that are placed in all museums: 'You can't touch!', which referring to a blind person is like telling a sighted person he cannot look. Out of this exasperation came my wife's idea, to collect in one place reproductions of the great masterpieces of art, so that even the blind could get to know them and enjoy the beauty of the masterpieces of human genius." [NetworkMuseum, 2017]

The project immediately found support from the municipality of Ancona and was established on May 29, 1993, with the contribution of the Marche Region. On November 25, 1999,

it was unanimously recognized as a state museum by the Italian Parliament, confirming its unique international significance.

2.1.2 The Omero Museum today

The museum has a collection of over 150 works, including sculptures, paintings, reliefs, and tactile reproductions of famous artworks [MinistryOfCulture, 2016]. It also organizes exhibitions, workshops, conferences, and educational activities for visitors of all ages and abilities.

The exhibition route is structured into rooms arranged in chronological order. It begins with the Greco-Roman room, followed by the Medieval/Renaissance room, and ends in the Contemporary room, which showcases numerous original pieces.

The complete collection is designed to be accessible and interactive for individuals with visual impairments. All the pieces displayed in the museum's rooms, which visitors can admire during their visits, are provided with comprehensive descriptive information in Braille, as well as in large print for those with low vision. Additionally, there are treadmills for exploring the higher parts of the sculptures.

However, the museum currently lacks the capability for visitors to digitally track their pathway and access additional information about the pieces on the fly.

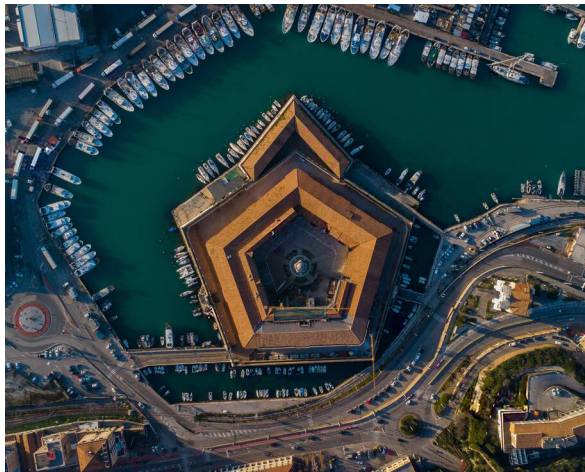


Figure 2.1: The Mole Vanvitelliana on which the museum is located today

The location of the museum has changed over time. At the beginning of its inauguration, it was located in three classrooms of the "Carlo Antognini" elementary school in Ancona; then, several years later, it was moved to a wing of Ancona's "Donatello" middle school. Today the museum occupies an entire side of the "Mole Vanvitelliana" (depicted in Figure 2.1), a pentagonal-shaped building dating from the first half of the 1700s by Luigi Vanvitelli. The building is located on an artificial island connected to the mainland by three bridges within the port of Ancona.

2.2 Internal organization

As a museum, the majority of customer interactions, such as ticket purchases, occur in person at the physical location. Nevertheless, the current website offers partial digitization by providing access to information regarding the museum's collection of artworks and past exhibitions. However, it lacks several functionalities like the possibility to remotely book visits by compiling a dedicated form. To determine the key features required for the development

of the application, it is essential to provide a concise overview of the museum's business logic.

2.2.1 Services offered

The museum offers three different kinds of service to clients, which are:

- guided and unguided tours for a fixed collection of works;
- guided and unguided tours for occasionally organized exhibitions;
- organized laboratory activities for groups of people with a minimum number of participants.

Effective communication with customers is vital in conducting museum activities. The reception is staffed with knowledgeable personnel who are available to assist customers by providing information about the services offered and guiding them through the ticket purchase and visit booking processes. To facilitate communication, the museum website displays contact information, including a telephone number and an email address. However, the museum currently does not offer a dedicated chat feature to address customer inquiries. The presence of such functionality would greatly benefit the museum, as it would encourage increased customer engagement and facilitate asynchronous message exchanges.

2.2.2 How a client visit takes place

Upon entering the museum, visitors are greeted at the reception area where they can purchase and validate tickets (which are shown in Figure 2.2). Generally, admission to the museum is free for both individuals and families, with the exception of exhibitions and laboratory activities that may have fees depending on the type of event. At the conclusion of their visit, visitors are encouraged to compile a paper survey, which collects anonymous data about the quality of their experience for statistical purposes.

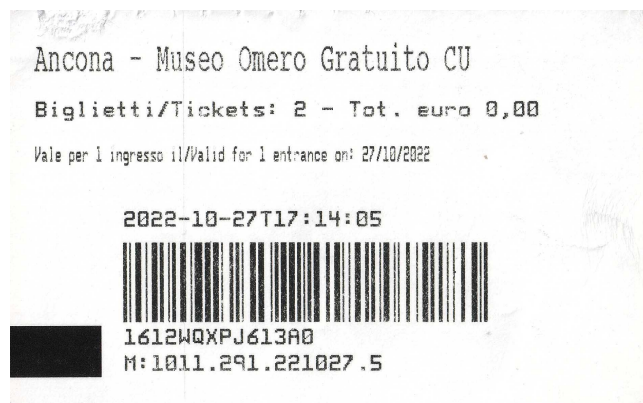


Figure 2.2: The Omero Museum access ticket

As stated in Section 2.2.1, clients can also request a guided tour in order to learn the details of the exhibited works and ask questions to the trained staff. The museum organizes clients who request a guide into groups to meet all the requests.

Clients have the option to make advance bookings to secure a spot for their visit. This can be done exclusively by telephone, calling the number provided on the museum's website. Reservations are required for laboratory services as the museum needs to ascertain the number of participants in order to properly organize the events.

2.2.3 The need for management software

The website provides visitors with easy access to all information pertaining to the museum's artworks and past exhibitions, allowing them to conveniently explore the content at their own pace. However, in terms of internal management, there is a notable absence of digitization. During interviews with the staff, it became clear that several tasks that could be automated are still carried out manually using basic tools like *Excel* sheets or paper notes.

Furthermore, the museum currently does not offer a mobile application to its clients. Although it primarily caters to the blind, it is essential to acknowledge that the majority of its clientele consists of individuals without significant visual impairments. Consequently, the development of a customer application would undoubtedly enhance the value of the museum's services, catering to a broader range of visitors.

As mentioned earlier, at the moment clients have only two options to book a ticket: either by making a phone call to reserve a spot or by visiting the physical museum location and interacting with the reception staff. In both cases, clients are unable to independently purchase a ticket for their visit without direct communication with the museum staff.

Currently, the museum relies on a *MySQL* database management system provided by the *WordPress CMS* to store all its data. This data is updated through an interface that executes low-level *MySQL* queries and is used to display information about artworks, rooms and exhibitions on the website.

Questionario di rilevazione del grado di soddisfazione dei visitatori

Gentile Visitatore,
grazie per averci fatto visita.
Le saremmo grati se volesse dedicare qualche minuto per rispondere al seguente questionario, che intende misurare il **grado di soddisfazione dei visitatori dei Musei Italiani**.
La sua opinione sarà utile per migliorare la qualità della visita e dei servizi offerti dai singoli Musei, ma anche l'immagine del Sistema Museale Nazionale nel suo insieme.
Tutte le risposte saranno trattate in forma anonima e nel rispetto della normativa sulla privacy.
La ringraziamo per la collaborazione.

NOME DELL'ISTITUTO CHE HA APPENA VISITATO [ogni istituto inserisca la propria denominazione]

Giorno della visita:
 domenica o altro giorno festivo
 sabato
 un giorno feriale

Modalità della visita: (è possibile selezionare più di una risposta)
 visita libera
 visita guidata
 partecipazione a un evento/manifestazione/conferenza
 altro (specificare) _____

È venuto con: (è possibile selezionare più di una risposta)
 da solo
 il partner/coniuge
 la famiglia
 amici/parenti/conoscenti
 una scolaresca
 un gruppo organizzato
 altro (specificare) _____

per partecipare a programmi e/o attività
 altro (specificare) _____

Rispetto alle sue esigenze, come giudica i seguenti aspetti della visita?
(Esprima la suo grado di apprezzamento su una scala da 1 a 5 oppure indichi perché non può rispondere)

	1	2	3	4	5	Non c'è	Non so	Non ho usufruito
Orari di visita								
Raggiungimento del luogo								
Informazioni di orientamento alla visita								
Percorso di visita								
Comunicazione contenuti (pannelli, didascalie, schede mobili, audioguida)								
Contenuti interattivi e multimediali (filmati, ricostruzioni virtuali, applicazioni scaricabili ecc.)								
Visita guidata								
Professionalità e cortesia del personale								
Pulizia e decoro								
Spazi di riposo/riflexione (poltrone, panchine ecc.)								
Servizi di ristorazione								
Punto vendita								
Altri servizi (specificare _____)								

La visita è stata all'altezza delle aspettative?
 E' stata decisamente migliore, sono entusiasta
 Mi sono sentito accolto non solo nel Museo, ma anche nel territorio che esso rappresenta
 E' stata una sorpresa positiva
 Sì, è stata come me l'aspettavo
 Pensavo meglio
 E' per appassionati/addetti ai lavori
 No, è stata una delusione
 altro (specificare) _____

Figure 2.3: The first two pages of the Omero Museum questionnaire

Another big problem is the compilation of questionnaires. The museum's current level of digitization reveals a big weakness in the compilation of questionnaires. The latter, spanning six pages, consist of subjective evaluation questions regarding the visitor's experience (as can be seen in Figure 6.20) and include biographical data, such as age and origin location. Unfortunately, the process remains entirely paper-based, with a limited number of printed copies placed on a table in the reception area. At the end of each workday, the museum staff faces the tedious task of manually transcribing the compiled responses into the computer system. These records are then collected and submitted to the administration at the end of each month for statistical purposes.

Bringing everything together, an ideal solution would involve a centralized system that

includes both an internal management application for the museum staff and a customer application. These would seamlessly interact with a shared data source, each having the necessary permissions. The management application would enable users to perform *CUD* operations on the database, while the customer application would retrieve and present the updated information. Additionally, the customer application would offer features such as storing purchased tickets and completed surveys and the possibility to communicate with the museum staff by sending messages in a dedicated chat.

2.2.4 Internal activities analysis

To facilitate the requirements collection and analysis, it is beneficial to visualize the sequence of activities involved in the typical interactions between customers and the museum. Figure 2.4 presents a *UML* activity diagram illustrating the key steps that a client must undertake to either purchase a ticket on-site or make a reservation through the website.

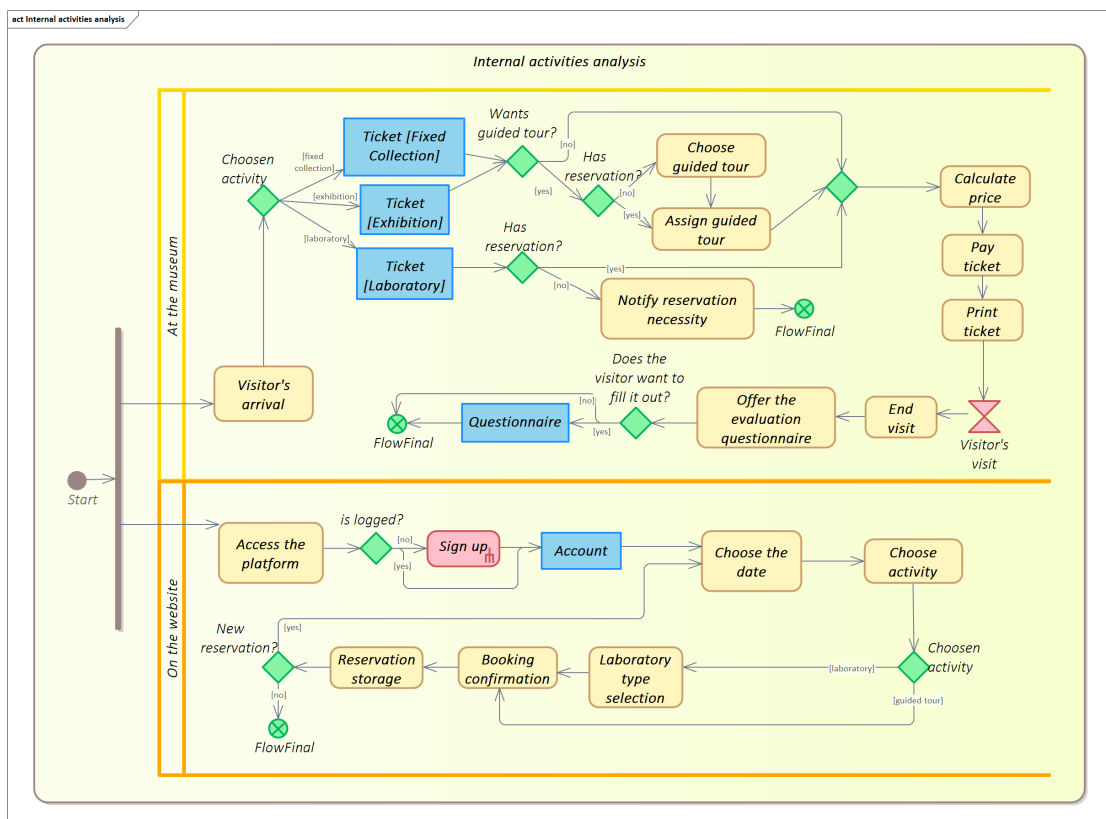


Figure 2.4: The activity diagram depicting the museum’s internal activities

The application that is to be developed should guarantee the seamless execution of these steps throughout the entire ticket-purchasing process.

Requirements Analysis

Having gained an understanding of the problem domain, as discussed in Chapter 2, we now move forward with organizing the raw information into requirements. These requirements are essential in meeting the expectations and needs of stakeholders.

In Section 3.1, we present a compilation of these requirements, enriched with detailed descriptions and priority indicators. Additionally, we provide a roster of non-functional requirements concerning the technologies to be used for the software's development.

Subsequently, in Section 3.2, we delve into the extraction of use cases from these functional requirements, commencing by identifying the actors that will interact with the system. We then present a list of these scenarios, each decorated with an elaborate description of their individual steps.

In Section 3.3, we demonstrate how the outlined use cases effectively address the corresponding functional requirements of the software using the mapping matrix.

Concluding this chapter in Section 3.4, we undertake an initial attempt to categorize the entities that have emerged from the requirements and use cases into distinct classes.

3.1 Application requirements

Before delving into the software architecture design, it is essential, in accordance with software engineering theory, to refine the information gathered from the interviews summarized in Chapter 2. This involves the organization of the data into functional and non-functional requirements.

Functional requirements hold explicit needs expressed by stakeholders and are essential for respecting the business logic. Conversely, non-functional requirements belong to more technical aspects, such as the adoption of specific technologies, which are more closely related to the solution domain rather than the problem domain.

As mentioned above, the software will consist of two front-end applications. One will be designed for mobile users, while the other will be designed for desktop machines located in the museum and operated by staff personnel.

Therefore, it makes sense to divide the functional requirements analysis into two separate sets, one for each application. However, since both applications will be connected to a shared data source and use the same technologies, the non-functional requirements will be the same for both.

3.1.1 Functional requirements

After conducting a thorough analysis, the functional requirements were gathered and organized in tables 3.1 and 3.2 based on the application they refer to.

They are prioritized using the *MoSCoW* method. This method provides a simple but effective means of organizing the development stages of software functionality. It is responsible for avoiding excessive efforts being devoted to less critical features, which could hinder the timely completion of more crucial functionalities within the specified time constraints. By prioritizing requirements based on their importance and impact, the developer can ensure that the most essential aspects of the project are addressed promptly, leading to a more effective and valuable solution for stakeholders.

Requirement	Description	Priority
FR-01 Artworks CRUD	The system will need to manage the CRUD operations of artworks, allowing staff users to create, read, update, and delete artwork records. Additionally, it will provide the option to sort the artwork list by title, creation date, author or storage room.	Must have
FR-02 Exhibits CRUD	The system will need to manage the CRUD operations of exhibits, allowing staff users to create, read, update, and delete exhibit records. Additionally, it will provide the option to sort the exhibits list by title, creation date, start date or end date.	Must have
FR-03 Tickets creation	The system will allow the staff users to create new ticket records for customers.	Must have
FR-04 Sold tickets index	The system will allow the staff users to visualize the list of printed tickets. Additionally, it will provide the option to sort the ticket list by validity date, purchase date, validation date or service type.	Must have
FR-05 Tickets validation	The system will allow the staff users to validate sold tickets.	Must have
FR-06 Questionnaires statistics	The system will need to manage statistics based on customer questionnaire completions. Additionally, it will provide a feature that allows staff users to specify a time range within which the statistics should be calculated.	Should have
FR-07 Customer communication	The system will allow the staff users to answer the customers' questions through an apposite chat.	Should have
FR-08 Staff account management	The system will allow the staff users to edit their account information. Additionally, it will provide the option to upload a custom profile avatar image.	Should have

Table 3.1: Functional requirements table for the internal management application

3.1.2 Non-functional requirements

Table 3.3 displays the non-functional requirements, which hold specific needs to be addressed during the later stages of software development and implementation. Among these, the most significant requirement is undoubtedly the utilization of the Microsoft framework

Requirement	Description	Priority
FR-09 Artworks index	The system will allow the customers to visualize the list of the museum's artworks. Additionally, it will provide the option to sort the artwork list by the room in which the artwork is exposed.	Must have
FR-10 Exhibits index	The system will allow the customers to visualize the list of the museum's past exhibits.	Must have
FR-11 Ticket purchase	The system will allow the customers to buy tickets for the requested service type.	Must have
FR-12 Bought tickets index	The system will allow the customers to visualize the list of bought tickets. Additionally, it will provide the option to sort the ticket list by validity date.	Must have
FR-13 Questionnaires compilation	The system will allow the customers to compile a questionnaire for each ticket the customer bought and validated. The questionnaire will contain several questions about the quality of the customer's visit.	Must have
FR-14 Personal statistics	The system will need to manage statistics based on the customer's bought tickets and compiled questionnaires.	Should have
FR-15 Customers chats	The system will allow the customers to request museum assistance through a dedicated chat. Additionally, it will provide the option to search inside past conversations.	Should have
FR-16 Customer account management	The system will allow the customers to edit their account information. Additionally, it will provide the option to upload a custom profile avatar image.	Should have

Table 3.2: Functional requirements table for customers' application

.NET MAUI, extensively discussed in Chapter 1. This framework serves as the foundation for the entire software development, providing the necessary tools and resources for creating a robust and cross-platform application.

3.2 Actors and use cases

After defining the functional requirements in Section 3.1.1, the next step in the analysis phase is to identify the scenarios that highlight the sequences of actions derived from the requirements. These scenarios, commonly referred to as "use cases," provide detailed descriptions of how the system will be interacted with.

To accomplish this, it is crucial to clarify the external actors who can interact with the system right from the beginning. These actors do not necessarily represent the individuals communicating with the system, but rather the roles they play in their interactions with it.

Requirement	Description
NFR-01 Credentials	The system will need to recognize the user by his/her credentials. Additionally, it will provide the option to register a new account for customers. The system will rely on the Firebase <i>Authentication</i> APIs to manage and store the account's credentials. It will also be able to remember the user's authentication status across different logins.
NFR-02 Database	The system will rely on the <i>Realtime Database</i> NOSQL DBMS to store all the software data. Furthermore, it will make use of <i>Web Socket</i> technology in order to be notified in real-time of external changes in the data of interest.
NFR-03 MAUI Framework	The system will be built on the .NET MAUI Microsoft's framework, leveraging its cross-platform capabilities to enable code sharing between the desktop devices for the staff application and the mobile devices for the customer application.
NFR-04 QR Code encoding	The system will rely on <i>QR code</i> encoding to enable remote ticket validation. For this purpose, it will also be able to scan and decode a <i>QR code</i> .
NFR-05 Dark theme support	The system will offer support for a dark theme option within the mobile application in order to reduce eye strain in low-light environments. Additionally, the staff application will feature the capability to toggle between various themes.
NFR-06 Charts support	The system will rely on Google's <i>Skia Graphics Engine</i> in order to render different types of charts.

Table 3.3: Non-functional requirements table

3.2.1 Actors

In our current context, the task of identifying actors is relatively straightforward. Considering the division of the software into two applications, namely the management and customer applications, the roles of external entities interacting with the system can be summarized by the following list:

- Staff;
- Registered staff;
- Customer;
- Registered customer.

The list is generated by the combination of two orthogonal levels of abstraction, namely the roles of the entities involved and their authentication level.

Regardless of the specific roles held by museum employees, their interactions with the system follow the same pattern. Therefore, they are collectively represented by the *Staff* and *Registered Staff* actors. The same applies to customers. They are all gathered under the *Customer* and *Registered customer* actors.

It is important to highlight that there exists a generalization dependence between the *Registered Staff* and *Staff* actors and between the *Registered Customer* and *Customer* actors. This implies that the *Registered Staff* and *Registered Customer* actors are specialized versions of their supertypes, enabling them to activate a broader range of use cases.

3.2.2 Use cases diagrams

For clarity, the use cases have been categorized into two distinct diagrams: one for the internal management application and another for the customer application.

The *extend* and *include* dependencies were employed between use cases to enhance the clarity of the diagrams, complying with the *UML* syntax. The *extend* dependency is applied when a use case is intended to enrich the sequence of another use case under specific conditions. As a result, the extending use case does not exist independently from the base use case and cannot be directly activated by an actor. Conversely, the *include* dependency is used to mitigate redundancy when certain use cases require invoking another use case's sequence within their own.

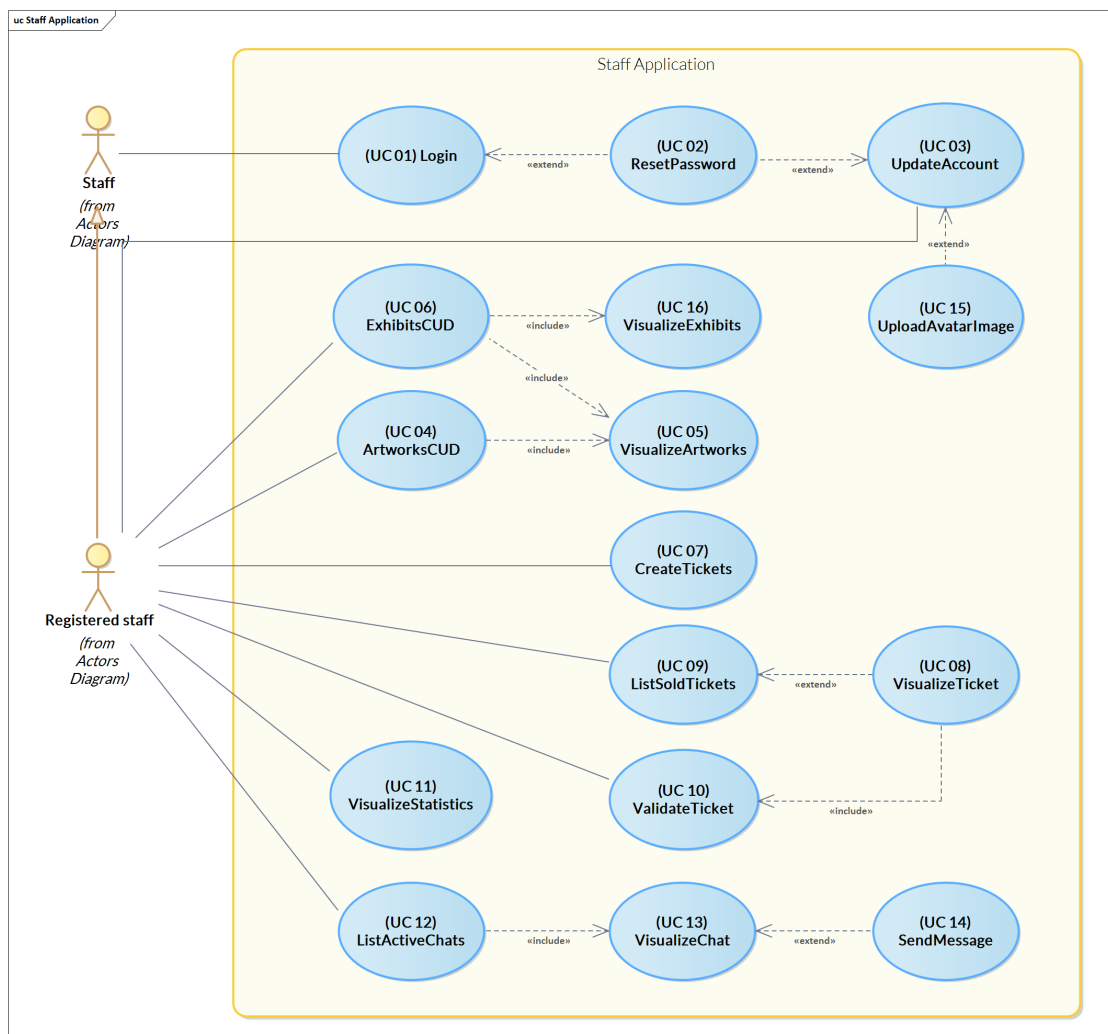


Figure 3.1: The use cases diagram of the staff application

The scenarios related to internal management are shown in Figure 3.1 and are analyzed in detail below. In the following, the term "main actor" will be used to refer to the actor or the actors that can activate the use case. The latter can be immediately determined by looking at the diagrams in Figures 3.1 and 3.2.

- **Login (UC 01):** this use case allows the authentication of an unauthenticated actor. The main actor provides their credentials, which the system then verifies. If the validation is successful, the system retrieves the account associated with the main actor and all

pertinent information. If there are any errors in the provided credentials, the system informs the main actor through an appropriate error message. If the main actor informs the system about the necessity to modify the password linked to their account, the system starts the *UC 02* sequence.

- ▶ **ResetPassword (UC 02):** this use case extends the sequence of the base one *UC 01* and allows the main actor to reset the password associated with their account. The system asks the main actor to input their email and then it sends a message to the provided mailbox containing instructions for changing the password.
- ▶ **UpdateAccount (UC 03):** this use case allows the main actor's account details to be updated. The system displays all the current information about the main actor's account, such as their name, surname and cellphone number. If the main actor is *Registered staff*, the system displays their role inside the museum too. The main actor makes any necessary changes and the system saves them. If the main actor informs the system about the necessity to modify the password linked to their account, the system starts the *UC 02* sequence. If the main actor informs the system about the necessity to modify the avatar image linked to their account, the system starts the *UC 15* sequence.
- ▶ **ArtworksCUD (UC 04):** this use case allows the main actor to manage the artwork catalogue. If the main actor intends to create a new artwork record, the system prompts them to input all relevant details, including the image file, title, author, exhibition room, materials, dimensions, creation technique, and textual description. Once the main actor successfully fills in all the necessary information, the system stores the new record and notifies them of the operation's success. Alternatively, if the main actor desires to update an existing artwork record, the system starts the *UC 05* sequence. The main actor selects the artwork they wish to modify. The system displays all pertinent information about the chosen artwork and offers the main actor the option to edit these details. After making the necessary changes, the main actor confirms, and the system saves the updates. Lastly, if the main actor intends to delete an existing artwork record, the system starts the *UC 05* sequence. The main actor selects the artwork for deletion and the system displays a confirmation prompt. Upon confirmation, the system removes the record as requested by the main actor.
- ▶ **VisualizeArtworks (UC 05):** this use case allows the main actor to visualize the artwork catalogue. The system presents a comprehensive list of currently stored artworks, featuring essential information such as title, author, registration date, and exhibition room for each entry. If the main actor is *Registered staff* and if they wish to arrange the catalogue based on a specific field, they communicate their preference to the system, which then proceeds to sort the artwork list according to the specified criteria. Upon selection of one of the listed artworks, the system shows all of the record's details beyond the already presented ones.
- ▶ **ExhibitsCUD (UC 06):** this use case allows the main actor to manage the exhibits' catalogue. If the main actor intends to create a new exhibit record, the system prompts them to input all relevant details, including the image file, title, start date, end date, and textual description. The system also asks to specify a list of artworks to be exposed by initiating the *UC 05* sequence. Once the main actor successfully fills in all the necessary information, the system stores the new record and notifies them of the operation's success. Alternatively, if the main actor desires to update an existing exhibit record, the system starts the *UC 16* sequence. The main actor selects the exhibit they wish to modify. The system displays all pertinent information about the chosen exhibit and offers the

main actor the option to edit these details. After making the necessary changes, the main actor confirms, and the system saves the updates. Lastly, if the main actor intends to delete an existing exhibit record, the system starts the *UC 16* sequence. The main actor selects the exhibit for deletion and the system displays a confirmation prompt. Upon confirmation, the system removes the record as requested by the main actor.

- ▶ **CreateTickets (UC 07):** this use case allows the main actor to buy a ticket. The system prompts the main actor to provide all necessary details, such as the desired service, validity day, presence of a guide service, and, if present, the scheduled visit time. Subsequently, the main actor fills in all required fields and confirms the action. Then, the system presents a summary of the purchase details, including the ticket price, and asks for confirmation from the main actor. If the main actor confirms the operation, the system manages the payment process and stores the newly generated ticket entry.
- ▶ **VisualizeTicket (UC 08):** this use case extends the sequence of the base one *UC 09* and *UC 20* and allows the main actor to visualize all the details of the selected ticket. The system displays comprehensive details of the ticket, including a QR code representation of its identifier. If the ticket is valid but has not yet been validated, and the main actor is the *Registered staff*, the system provides them with the choice to validate it. If the main actor chooses to proceed, the system starts the *UC 10* sequence.
- ▶ **ListSoldTickets (UC 09):** this use case allows the main actor to visualize the list of bought tickets. The system presents a comprehensive list of bought tickets, featuring essential information, such as validity date, bought date, and validation date for each entry. If the main actor is *Registered customer*, the system applies a filter to display only tickets bought by that customer. If the main actor is *Registered staff* and if they wish to arrange the list based on a specific field, they communicate their preference to the system, which then proceeds to sort the ticket list according to the specified criteria. If the main actor selects one of the listed tickets, the system starts the *UC 08* sequence.
- ▶ **ValidateTicket (UC 10):** this use case allows the main actor to validate a valid ticket. If the system is not currently displaying a ticket, it activates the device's camera to scan a QR code. The main actor positions the ticket they intend to validate in front of their device's camera. The system exploits the provided ID to access all the ticket information. If the ticket is both valid and has not been validated previously, the system proceeds to validate it, records the validation date, and informs the primary actor of the operation's success. Conversely, if certain conditions are not met, the system notifies the main actor of the error.
- ▶ **VisualizeStatistics (UC 11):** this use case allows the main actor to visualize the statistics calculated on the basis of customers' questionnaire completions. The system gathers data from questionnaire completions over the past three months and generates corresponding charts for key metrics, such as average ratings, visit types, companions, visit purposes, study titles, intent to return, and completion ratios. Furthermore, the system collects data on ticket sales from the last three months and computes the average number of weekly ticket purchases. If the main actor modifies the time range, the system fetches the necessary data and recalculates the statistics accordingly.
- ▶ **ListActiveChats (UC 12):** this use case allows the main actor to visualize the list of currently active chats. The system gathers records of users who have sent messages to the museum and presents them in a sorted list based on the date of their most recent message. For each of them, the system shows their avatar image, name, surname and

last access date. If the main actor chooses a specific chat from the list, the system starts the *UC 13* sequence.

- ▶ **VisualizeChat (UC 13)**: this use case allows the main actor to visualize a chat along with all the messages that have been exchanged within it. The system fetches all messages from a specified chat and presents them in a sorted list according to the message date. Each sent message is then accompanied by a suitable icon indicating whether the message has been delivered or read. If a new message is sent in the current chat, the system fetches and updates the chat's message list accordingly. If the main actor wants to send a message, the system starts the *UC 14* sequence.
- ▶ **SendMessage (UC 14)**: this use case extends the sequence of the base one *UC 13* and allows the main actor to send a message on a given chat. The main actor composes the message they wish to send in the chat and confirms the action. The system then fetches the message, records the sending date, and stores it. Finally, the system updates the list of messages in the current chat.
- ▶ **UploadAvatarImage (UC 15)**: this use case extends the sequence of the base one *UC 03* and allows the main user to upload their account's avatar image. The system prompts the primary actor to select an image file from their device. The primary actor then locates the desired image file and confirms the action. If the file size remains below a specific limit, the system proceeds to store the image and associates it with the account's avatar image. However, if the file size exceeds the specified limit, an error message is displayed by the system.
- ▶ **VisualizeExhibits (UC 16)**: this use case allows the main actor to visualize the exhibits' catalogue. The system presents a comprehensive list of currently stored exhibits, featuring essential information such as title, start date, end date, and registration date for each entry. If the main actor is *Registered staff* and if they wish to arrange the catalogue based on a specific field, they communicate their preference to the system, which then proceeds to sort the artwork list according to the specified criteria. Upon selection of one of the listed exhibits, the system shows all of the record's details beyond the already presented ones.

The scenarios related to the customer application are shown in Figure 3.2 and are analyzed in detail below. The diagram employs a distinct style for use cases that are shared with the diagram presented in Figure 3.1. These use cases are depicted in a grey colour, signifying their nature as links to use cases defined elsewhere rather than being standalone use cases. Therefore, the description of such use cases is not repeated here and can be found above.

- ▶ **SignUp (UC 17)**: this use case allows the main actor to register a new account. The system prompts the main actor to provide essential details for setting up a new account, including email, username, password, and password confirmation. Additionally, optional fields, like name, surname, and cell phone number can be filled in. Upon submission of all necessary information, if the provided email is not already associated with an existing account, and the password matches the password confirmation, the system proceeds to create and store a new profile record. After this, it starts the *UC 01* sequence for logging into the newly created account. In case any of the preceding conditions are not fulfilled, the system presents an error message.
- ▶ **CompileQuestionnaire (UC 18)**: this use case allows the main actor to fill out an evaluation questionnaire regarding a conducted visit. The system prompts the primary actor with evaluation questions concerning their experience, including aspects like

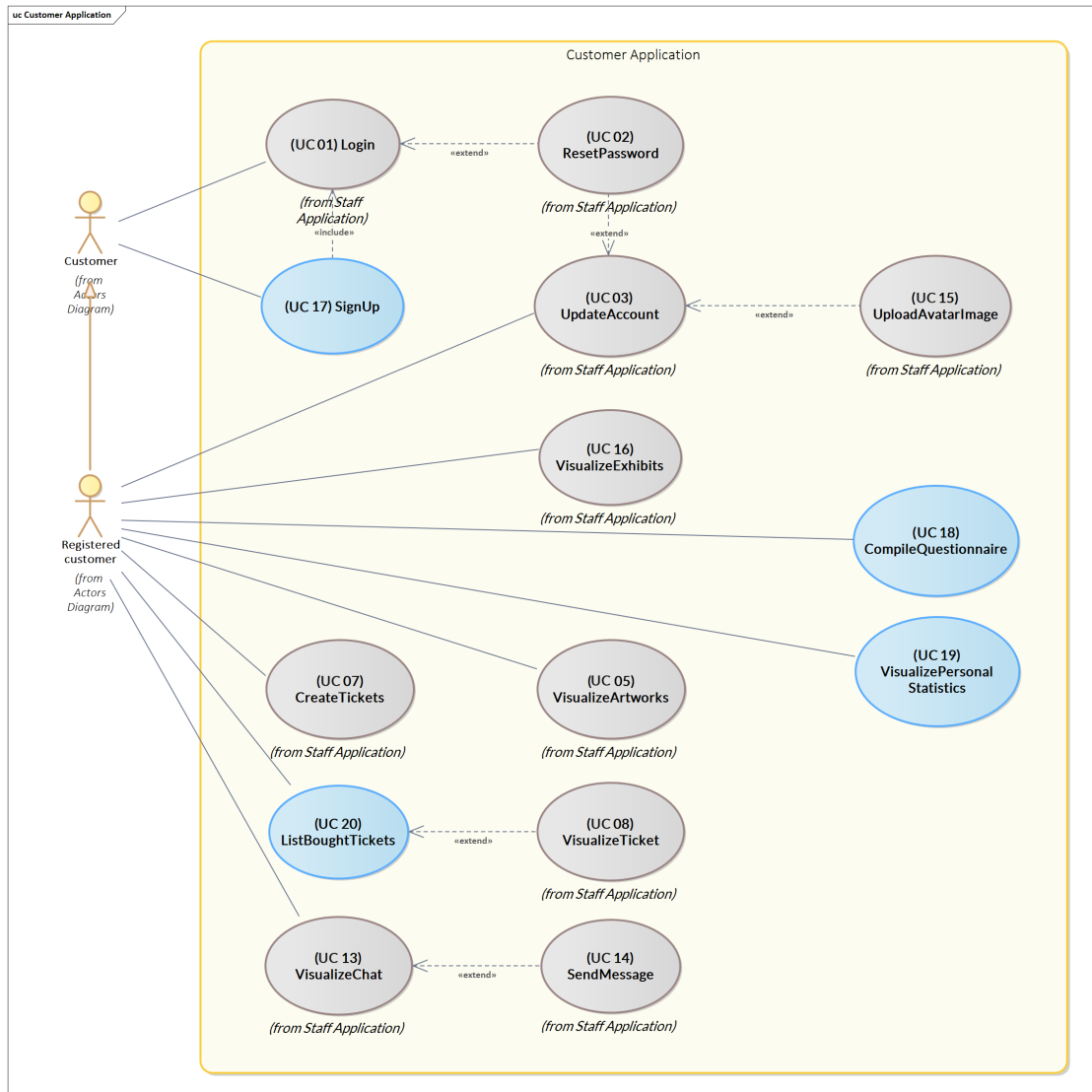


Figure 3.2: The use cases diagram of the customer application

the presence of companions, motivation, intention to revisit, evaluation of the visit, overall experience assessment, and assessment of the museum’s facilities. The main actor proceeds to complete all the required fields and confirms the action. Finally, the system stores the newly generated record and presents a suitable message.

- **VisualizePersonalStatistics (UC 19):** this use case allows the main actor to visualize their personal statistics. The system retrieves all the ticket records associated with the currently logged-in account. Then, it processes the data to generate statistics, including the ratio of visits to completed questionnaires and the types of services utilized.
- **ListBoughtTickets (UC 20):** this use case allows the main actor to visualize the list of their bought tickets. The system gathers the records of the tickets and presents them in a sorted list based on the date of their purchase. For each of them, the system shows their validity date, validation state and service type. If the main actor chooses a specific ticket from the list, the system starts the *UC 08* sequence.

3.3 Mapping matrix

The mapping matrix serves as a valuable tool for ensuring that all requirements that emerged from stakeholders’ needs are addressed by the proposed use cases. Nevertheless, the established relationship is not one-to-one; certain requirements may be catered to by multiple use cases. Given the division of functional requirements into those for internal management and those for customers, it can occur that certain requirements referring to the same interaction with the system, but associated with distinct actors, are fulfilled by a single use case.

Figure 3.3 illustrates the mapping matrix, depicting use cases along the rows and functional requirements along the columns. Whenever there is a dependency between a use case and a requirement, a symbol is indicated in the corresponding cell.

	FR-01 Artworks CRUD	FR-02 Exhibits CRUD	FR-03 Tickets creation	FR-04 Sold tickets index	FR-05 Tickets validation	FR-06 Questionnaires statistics	FR-07 Customer communication	FR-08 Staff account management	FR-09 Artworks index	FR-10 Exhibits index	FR-11 Ticket purchase	FR-12 Bought tickets index	FR-13 Questionnaires compilation	FR-14 Personal statistics	FR-15 Customers chats	FR-16 Customer account management
(CU 01) Login								↑								↑
(CU 02) ResetPassword								↑								↑
(CU 03) UpdateAccount								↑								↑
(CU 04) ArtworksCUD	↑															
(CU 05) VisualizeArtworks	↑								↑							
(CU 06) ExhibitsCUD		↑														
(CU 07) CreateTickets			↑								↑					
(CU 08) VisualizeTicket				↑												
(CU 09) ListSoldTickets				↑												
(CU 10) ValidateTicket					↑											
(CU 11) VisualizeStatistics						↑										
(CU 12) ListActiveChats							↑								↑	
(CU 13) VisualizeChat							↑								↑	
(CU 14) SendMessage							↑								↑	
(CU 15) UploadAvatarImage								↑								↑
(CU 16) VisualizeExhibits		↑								↑						
(CU 17) SignUp																↑
(CU 18) CompileQuestionnaire												↑				
(CU 19) VisualizePersonal Statistics													↑			
(CU 20) ListBoughtTickets											↑					

Figure 3.3: The mapping matrix

3.4 Class diagram

The following is an initial attempt to transition into an object-oriented model. Given the current state of analysis, the focus will remain on higher-level aspects, avoiding the complexities related to software implementation. The classes that will be highlighted are those representing the data model that the application will have to manage.

Following the principles of database design theory, the requisites of the data that the system needs to store will be refined for each entity outlined within the business logic.

3.4.1 Data requirements

Based on the refined requirements introduced at the beginning of this chapter, it was possible to highlight the data component requirements. Primarily, entities stemming from the problem domain encompass artworks, exhibits, tickets, questionnaires, customers, staff employees, and chats, along with their corresponding messages. For each of these entities, the system is required to store specific information, which can be detailed as follows:

- In relation to artworks, there's a necessity to capture data elements, such as the title, author, registration date, physical dimensions, image representation, exhibition room, material composition, modelling techniques employed, and a succinct description.
- With regard to the exhibits, there's a necessity to store information about the title, registration date, start date, end date, picture, description and the list of showcased artworks.
- Regarding tickets, the system must retain specific information including the purchase date, validity period, validation date, requested service type, and potential inclusion of a guide. Additionally, the system must record a distinct identifier for each ticket to permit recognition through the scanning of its associated QR code.
- Turning our attention to questionnaires, it is essential for the system to retain information such as the date of the visit, the type of visit, companions, motivation, study title, intention to revisit, visit evaluation, experience evaluation, structure evaluation, and the completion date.
- Concerning customers' accounts, the system should store particulars such as the username, first name, last name, cellphone number, profile picture, list of purchased tickets, list of completed questionnaires, and the messages exchanged with the museum.
- Regarding the employees' accounts, the system needs to memorize details about the first and last name, cellphone number, email and role within the museum.
- In relation to chats, it is important to store the list of message exchanges, both those sent by the customer and those sent by the museum.
- In conclusion, for each message, the system should store information about the time of sending, the textual content, and whether it has been read by the receiver.

3.4.2 Translation towards the object-oriented model

Bringing everything together, a UML class diagram satisfying all the aforementioned data requirements listed in Section 3.4.1 has been constructed. This diagram can be located in Figure 4.1.

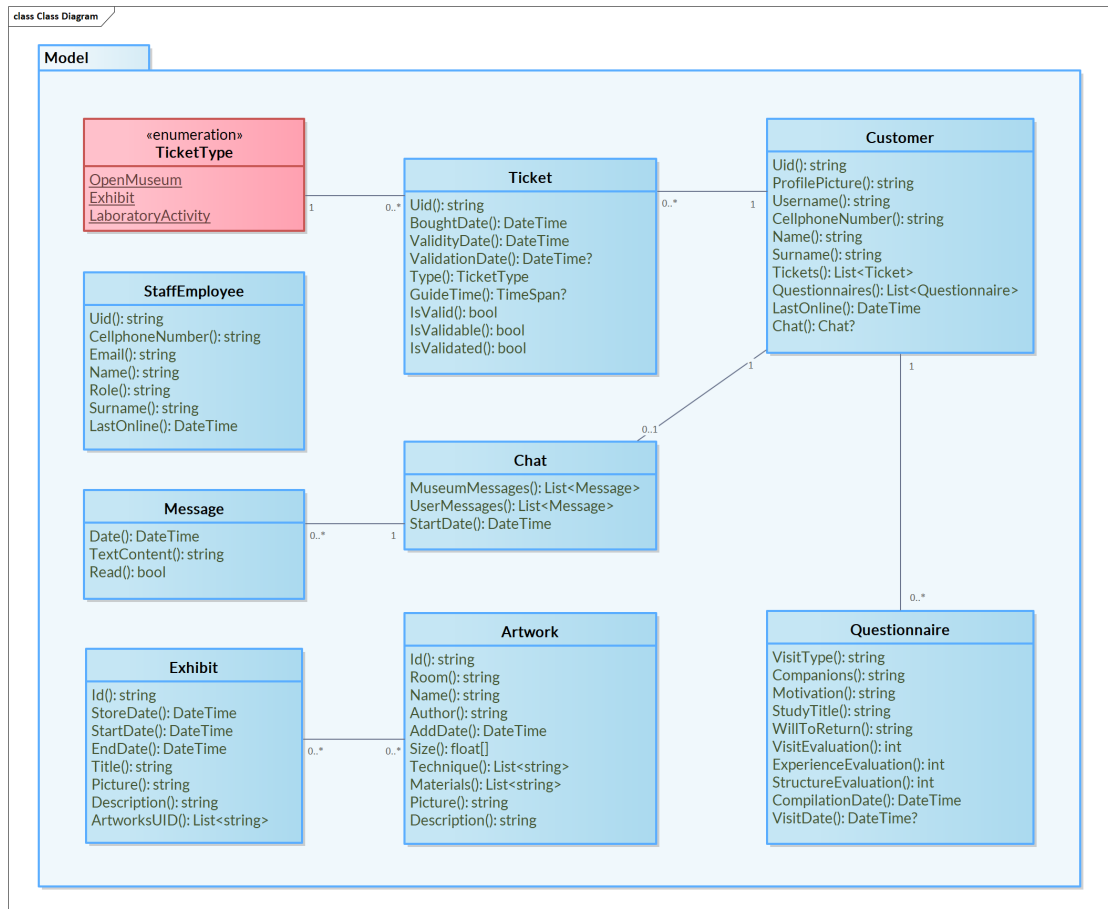


Figure 3.4: The analysis class diagram

The diagram is concise but effectively fulfils all the stipulated data requirements. Notably, certain attributes within the classes employ non-primitive types. Given the clear intent to adopt C# for software development, classes like `DateTime`, `TimeSpan` and `List<T>` are used as types. However, this does not undermine the language-independent nature of design diagrams, as these classes remain universally available across languages.

Additionally, the use of *nullable types* is notable, indicated by appending the "?" symbol to the attribute type. While necessary for programming languages like C# that support *null safety*, this element can be disregarded by languages that do not support it.

In the previous chapter, significant decisions were made regarding the organization of upper-level software components. In this chapter, we will further refine these choices in a pragmatic manner by aligning them with the previously stated non-functional requirements, namely the technologies to be used.

Following the presentation of appropriate justifications, we proceed in Section 4.1 to design the database's structure. In this context, we show how the framework can be naturally derived from the previously delineated analysis class diagram.

We continue in Section 4.2, wherein we delve into the design of the screens for both applications and delineate the navigation between them. Then, after providing the visual maps of the views, we proceed with the exposition of their mockups, namely their graphic prototypes.

In Section 4.3, we outline the inherent structure of the software, presenting a comprehensive overview of the requisite view and view model classes essential for realizing the screens as previously defined.

We conclude this chapter in Sections 4.4 and 4.5, presenting UML sequence and activity diagrams that pertain to the most noteworthy use cases.

4.1 Database structure

Considering the arrangement of the model classes outlined in Section 3.4.2, the next step involves designing the database structure responsible for storing instances of these classes.

As discussed in Section 3.1.2, the software will utilize Firebase's *Realtime Database* service. Thus, it becomes crucial to grasp the operational mechanisms of this database management system before proceeding with its structural design.

4.1.1 Firebase Realtime Database

The Firebase *Realtime Database* has emerged as a pivotal innovation in the realm of modern data storage. Along with *Firestore*, these are the two solutions offered by Google Firebase for data management. Its distinctive architecture underscores a shift towards a non-relational structure, redefining the paradigms of real-time data management. This novel approach allows for seamless synchronization of data across multiple clients and real-time updates.

Unlike traditional relational databases, *Realtime Database's* schemaless structure is extremely simple. By embracing a *JSON*-like tree structure, it facilitates efficient data organization and retrieval, while also mitigating the complexities associated with schema alterations. Data branches off from a central node known as the *root*. Adhering to the non-positional notation inherent in the *JSON* format, each data element is presented as a key-value pair, with the value having the potential to signify a primitive type like a string, number, or boolean, or alternatively, a collection of further key-value pairs.

Being a *NoSQL DBMS* and lacking a fixed schema, the need for traditional database design steps is obviated. Conversely, the development adheres to an "agile" approach, focused on structuring data, while also possibly making use of redundancy, to minimize the number of remote database queries. This strategy aims to aggregate data involved in operations that would have used *join* operators in a relational database, which is infeasible in this non-relational context.

This approach indeed comes with its own set of drawbacks. While faster and more adaptable than the traditional method, it unavoidably incorporates redundancy, as mentioned earlier. This redundancy introduces functional dependencies among data, potentially leading to anomalies. Given the absence of standard forms that typically ensure schema quality, developers must exercise caution during CRUD operations on the database. It becomes essential to account for all instances of redundancy that might be impacted when editing specific data.

Within the scope of the current application, the *Realtime Database* proves to be an optimal choice. In fact, there's no need to manage a substantial volume of data, and intricate queries or transactions are not essential. What is important is the ability to swiftly access small data sets, often in real-time, especially when external alterations occur.

4.1.2 Nodes design

After understanding the elementary structure of the database, listing the nodes that should reside directly beneath the root becomes a necessary task. By examining the aggregation dependencies among the model entities highlighted in Section 3.4, and taking into account the operational requirements outlined in Section 3.2.2, it becomes evident to propose the subsequent list of nodes:

- Staff employees;
- Customers;
- Exhibits;
- Artworks.

Each node contains a list of instances of the corresponding class.

The overall database structure is depicted in Figure 4.1, where each node contains a single instance for clarity.

Supported by the composition dependency involving mandatory customer participation, instances of tickets, questionnaires, and chats do not own an independent node, but are logically organized beneath the *Customers* node. The same applies to instances of messages, which are always associated with the respective chats to which they belong.

In contrast, instances of artworks, despite being aggregated within exhibit instances, exist independently from them. Consequently, artworks occupy a distinct node, avoiding the need to search for them within the instances of exhibits. Instead, accessing the list of artworks is streamlined by directly querying the *Artworks* node. On the other hand, when dealing with an exhibit instance, it is possible to obtain the list of displayed artworks by referencing its artworks identifiers list and retrieving the needed instances from the *Artworks* node.

4.2 Applications screen design

The following sections deal with the navigation design for both applications. This will provide a clear roadmap for the implementation phase, guiding the screens coding process.

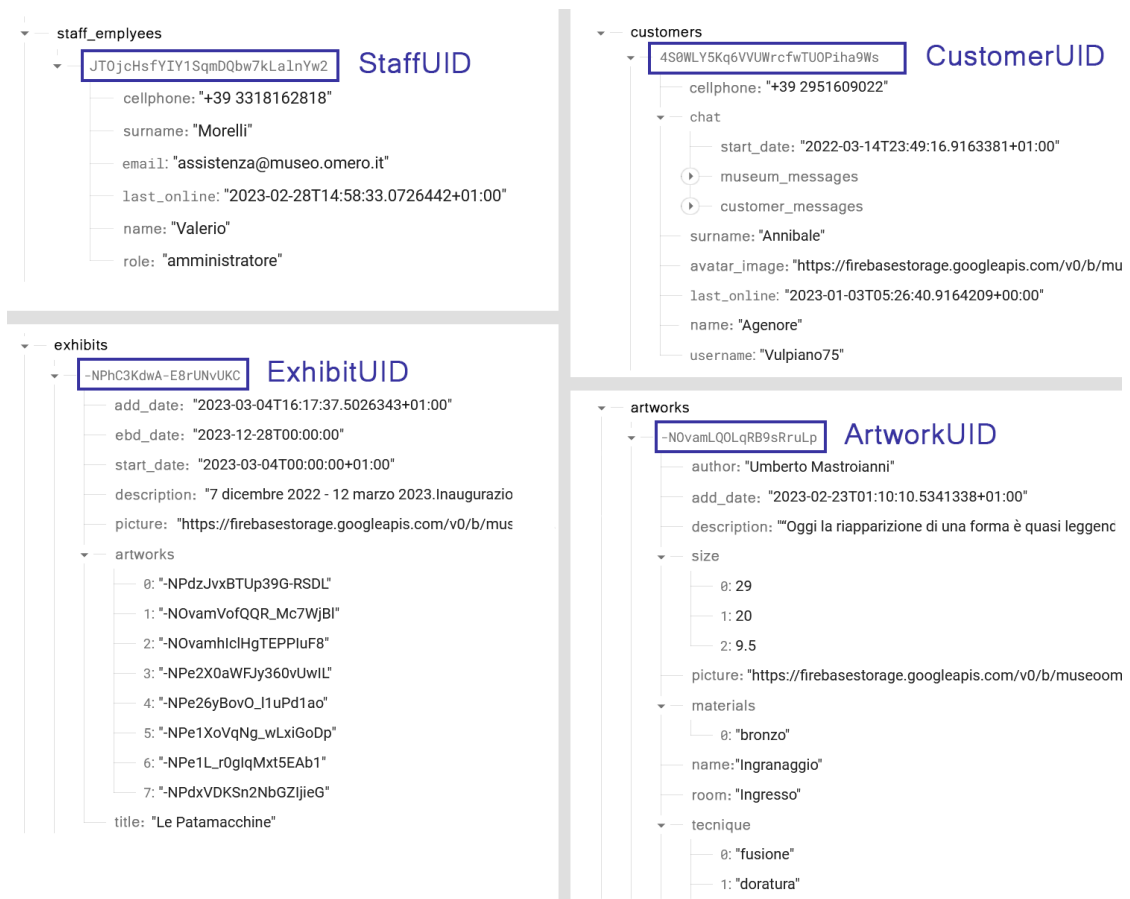


Figure 4.1: The *Realtime Database DBMS* overall structure

4.2.1 Applications map

To meet the stipulated requirements, it is essential to create the necessary screens for both applications. Each screen will address a subset of the functional requirements. Collectively, these screens should be interconnected in an intuitively seamless manner, ensuring an adequate user experience and minimizing the time required for users to access the information they are interested in.

For the customer application, navigation between pages accessible to the *Registered customer* role will be achieved through a *bottom bar*, a conventional graphical navigation element often used in mobile applications. On the other hand, the staff application will incorporate a *flyout shell*, which is an expandable side menu.

Although initially devised for web page design, visual maps have demonstrated their versatility in illustrating screen navigation for various applications. These maps offer a visual depiction of individual screens, interconnected to form the navigational structure of the application. The screens are arranged in a hierarchical tree and are adorned with a list of macro components they should contain. Several of these components are further enhanced with basic wireframes, indicating their intended graphical aspect.

Figures 4.2 and 4.3 showcase the visual maps for both applications. The initial screen is the *Login* in both cases. It is worth observing that this does not imply the necessity for users to input their credentials every time they launch the application. In fact, as outlined in *NFR-01* in Section 3.1.2, the software will retain its authentication status across multiple logins.

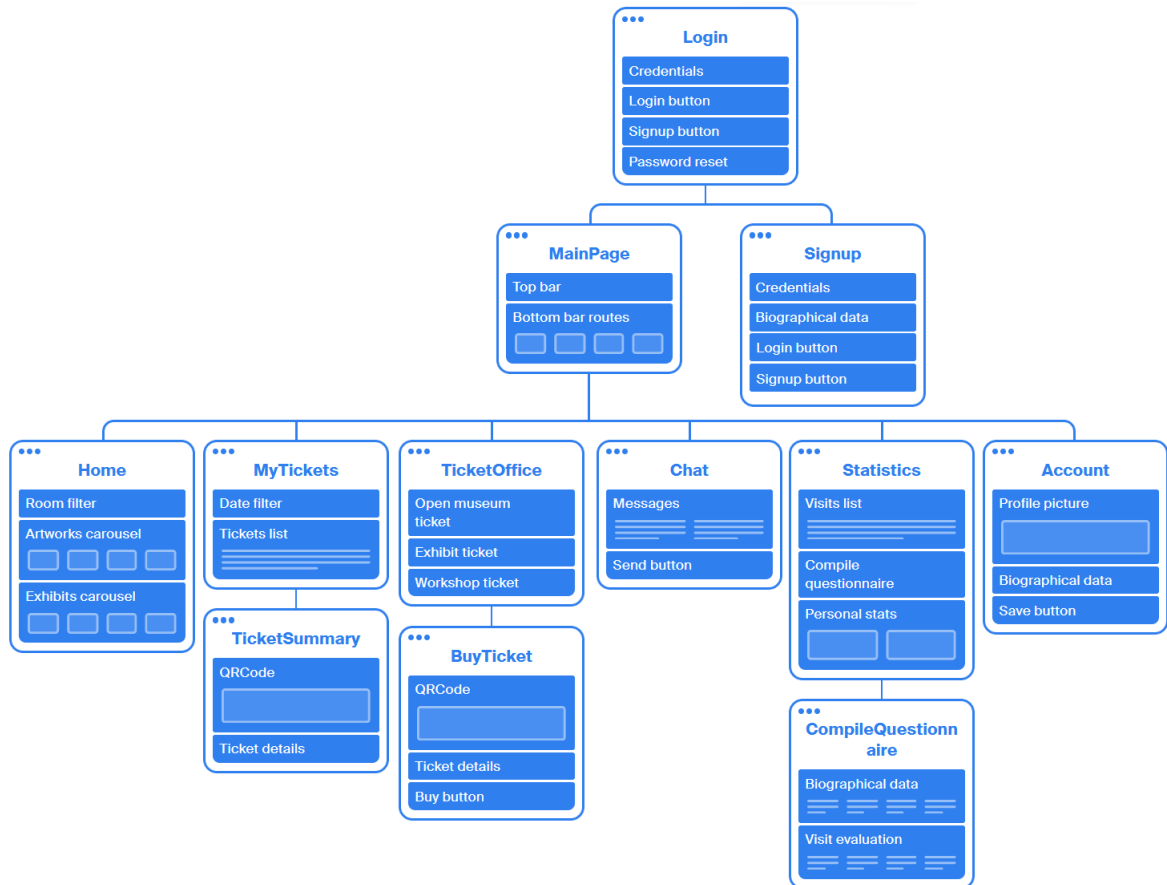


Figure 4.2: The screens map for the customer application

4.2.2 Mockups

Mockups are visual representations or prototypes that provide a tangible and often simplified preview of a design concept or an entire screen, typically used in the context of software development. They are generally much easier to develop than the actual views implementation and so serve as a crucial tool for refining and validating design ideas, allowing for early feedback and adjustments before the actual development process begins.

In this context, the detail in the mockups is intentionally maintained at a relatively high level due to the capabilities of the utilized framework, which enables the creation of visually appealing graphical interfaces with minimal exertion. Across both applications' screens, as explained in the preceding section, the *flyout shell* and *bottom bar* navigation components are recurrent elements. The former appears as a black side menu within the staff application, while the latter manifests as a list of icons situated at the lower part of the screens in the customer application.

Lastly, a crucial point to highlight is that mockups have been developed in the Italian language. This decision aligns with the will of stakeholders, as Italian serves as the primary language of interaction with the software's users. In order to enhance interoperability, it becomes imperative to centralize the strings within the code. This approach effectively abstracts the linguistic dependency of the software and empowers it to flexibly adopt different presentation languages from the list of available translations.

In Figures 4.4 - 4.9 the mockups related to the staff application are shown, while, in Figures 4.10 - 4.12 those related to the customer application are reported.

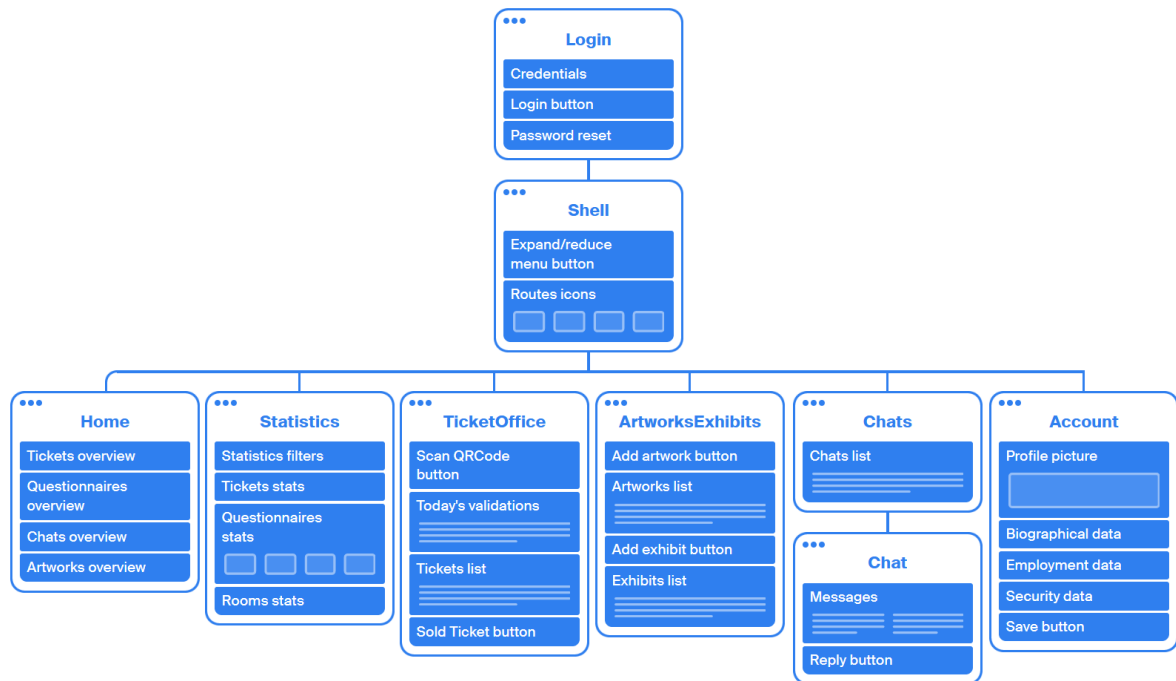


Figure 4.3: The screens map for the staff application

4.3 Refined class diagram

The software design process progresses with a more detailed refinement of the class diagram introduced in Section 3.4. The classes presented below do, in fact, consider features of software implementation.

4.3.1 The MVVM as the reference architectural pattern

As accurately explained in Section 1.4.1, the adoption of the *MVVM* architectural pattern is the preferred choice of the majority of modern frameworks, including, of course, *.NET MAUI*. This pattern delineates the organization of classes into three distinct components: *model*, *view*, and *view model*. Hence, it is entirely fitting that classes have been categorized into three homonymous packages.

In terms of pattern variants, the preference leans towards the *view-first composition*, due to its simplicity. Thus, the view classes will maintain references to their corresponding view model instances. As previously clarified in Section 1.2.4, the management of these instances will be encapsulated and entrusted to MAUI's dependency injection container. This, of course, requires the proper registration of the relevant classes within the container.

It is important to note that across the two applications to be built, the only components that are needed to change are screens. The model class, in fact, as well as the database repository, are shared across both of them. However, the same cannot be said for the view and view model classes, which differ, barring small graphical components, between the two applications. Hence, even though *.NET MAUI* is a cross-platform framework, capable of rendering a single view across different devices, in this context, it becomes a necessity for the views to be shown conditionally based on whether the application is executing on a mobile or desktop device.

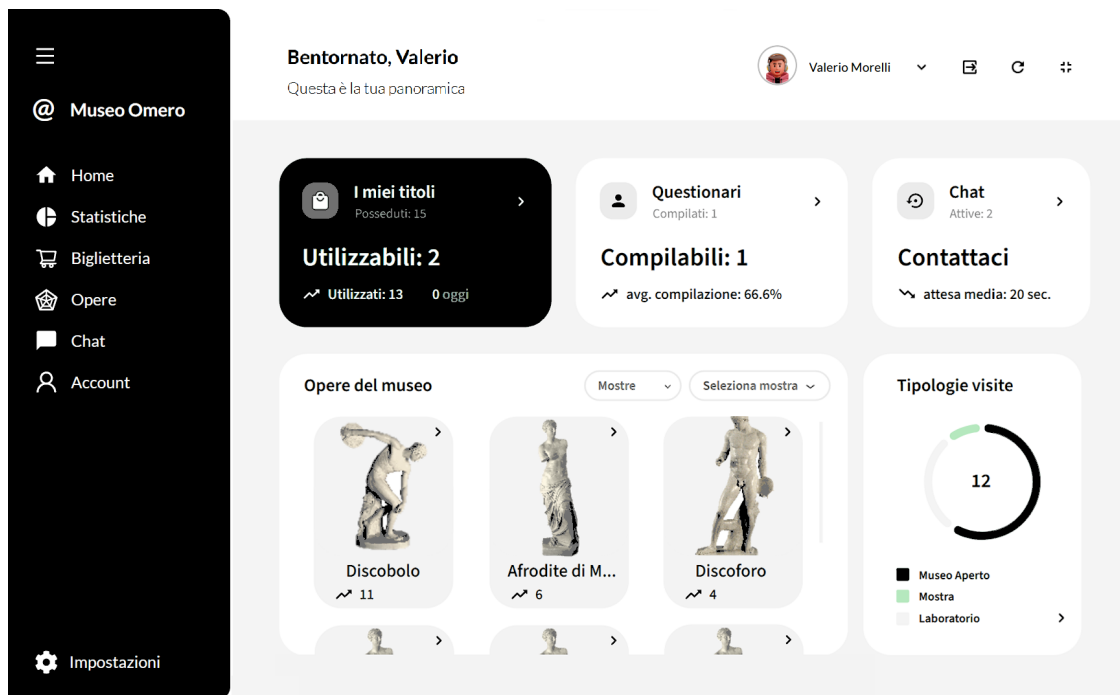


Figure 4.4: The mockup for the staff application’s home page

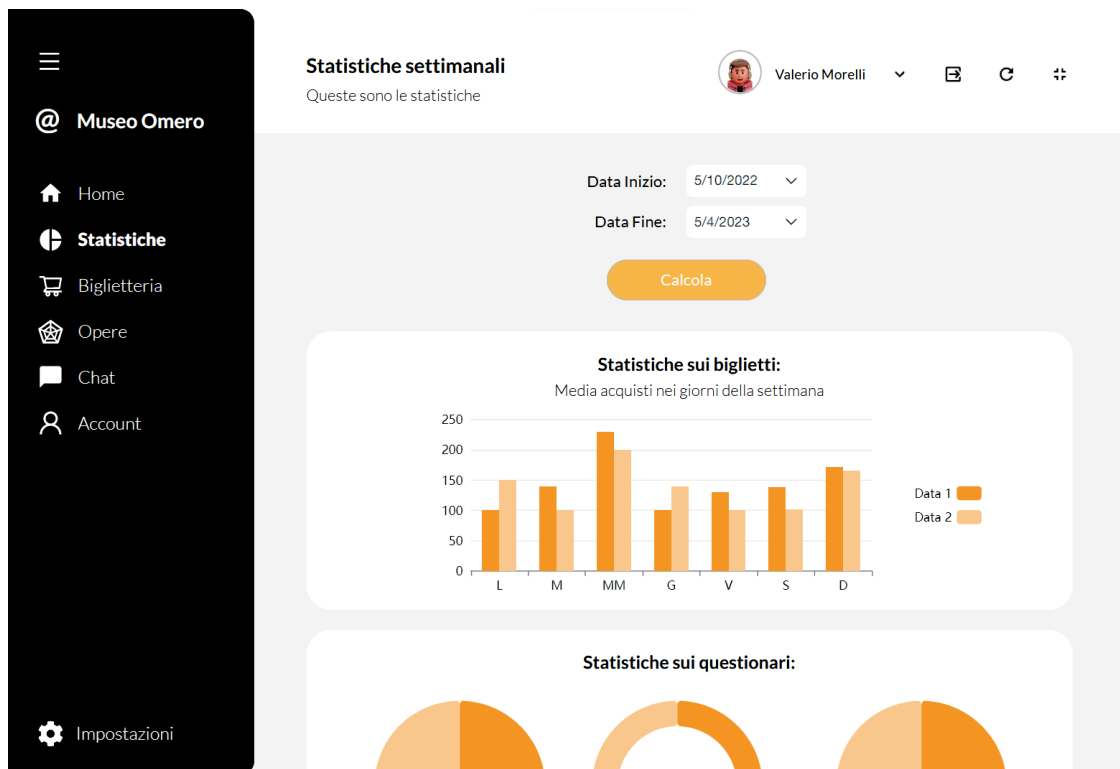


Figure 4.5: The mockup for the staff application’s statistics page

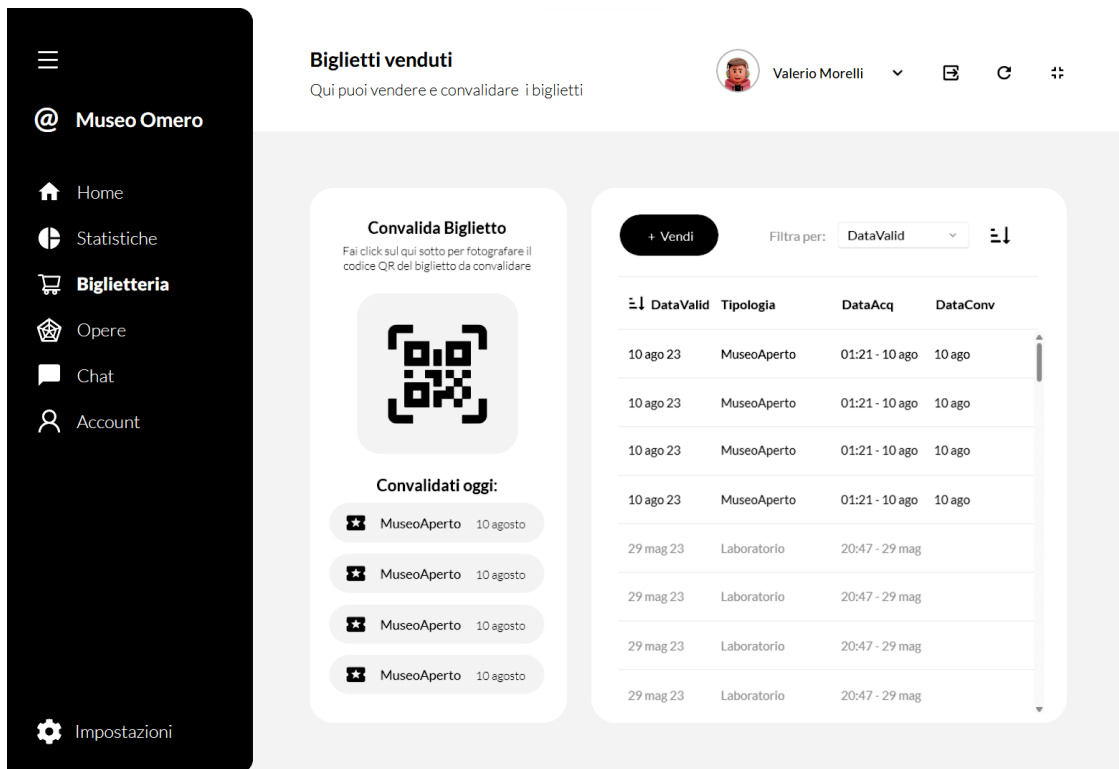


Figure 4.6: The mockup for the staff application’s ticket office page

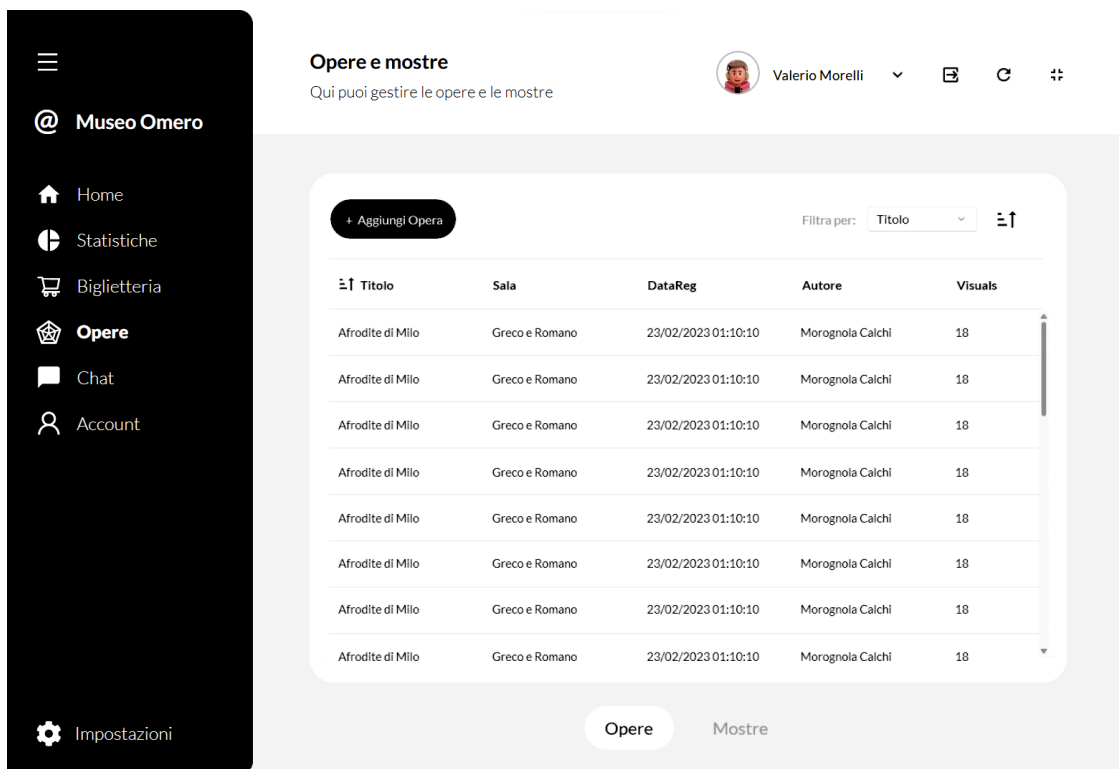


Figure 4.7: The mockup for the staff application’s artworks page

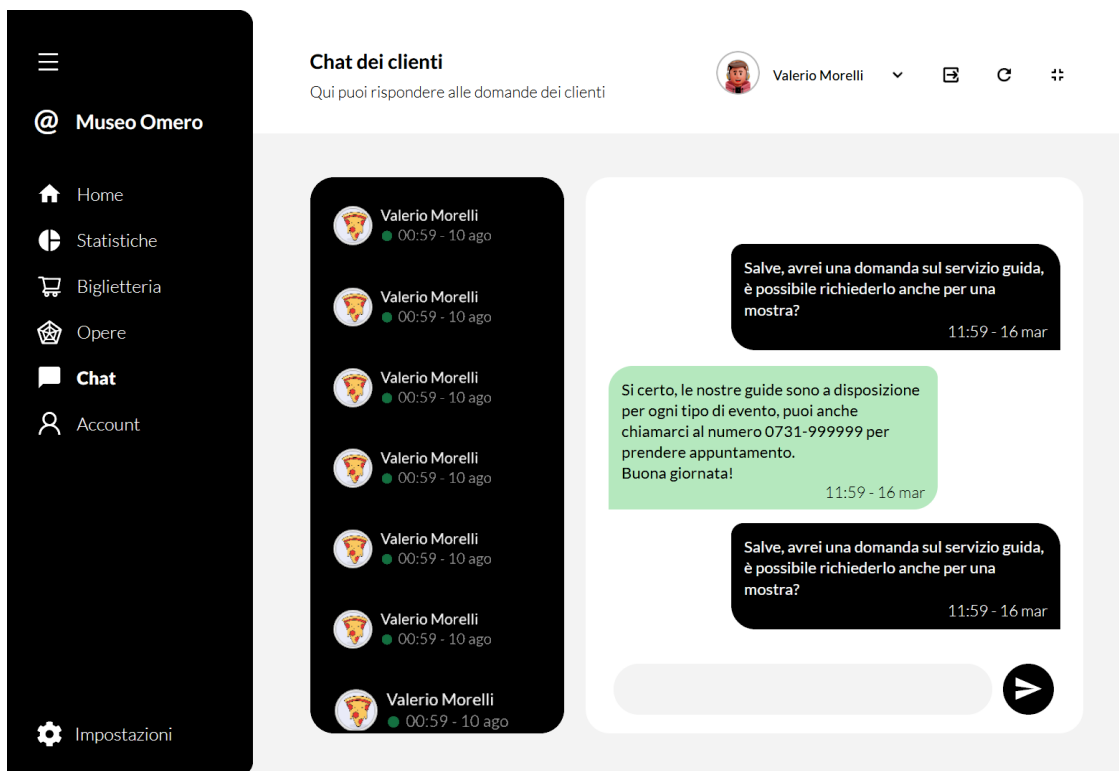


Figure 4.8: The mockup for the staff application’s chats page

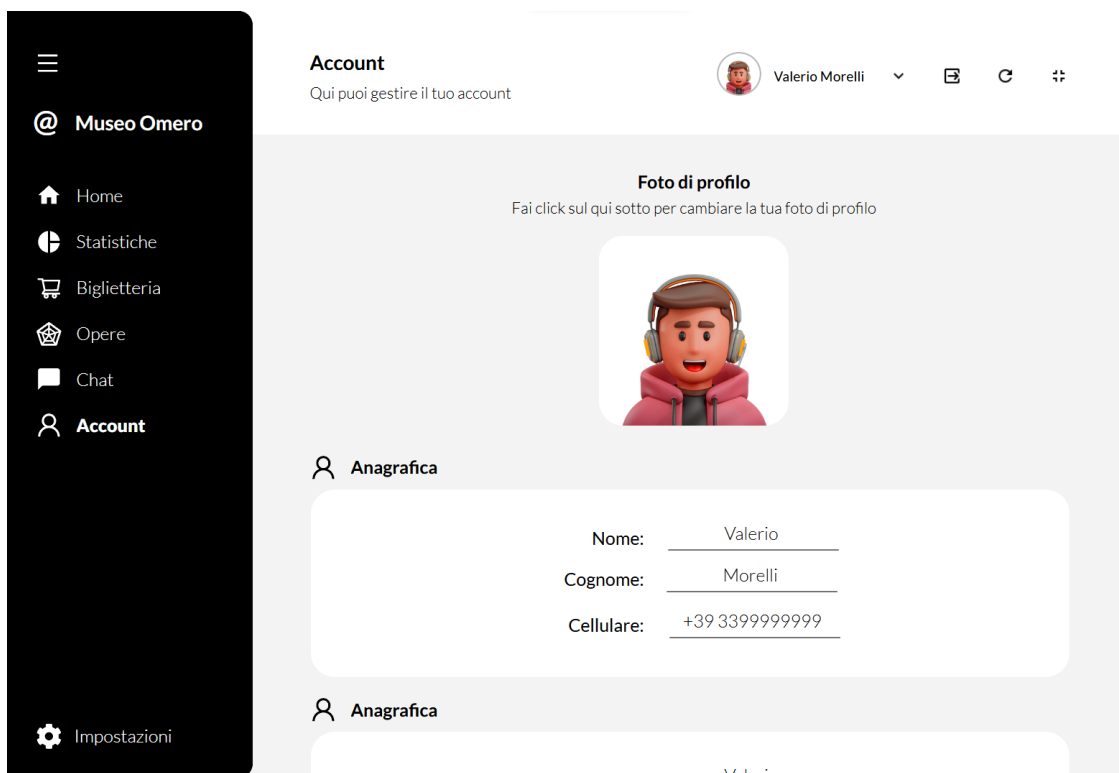


Figure 4.9: The mockup for the staff application’s account page

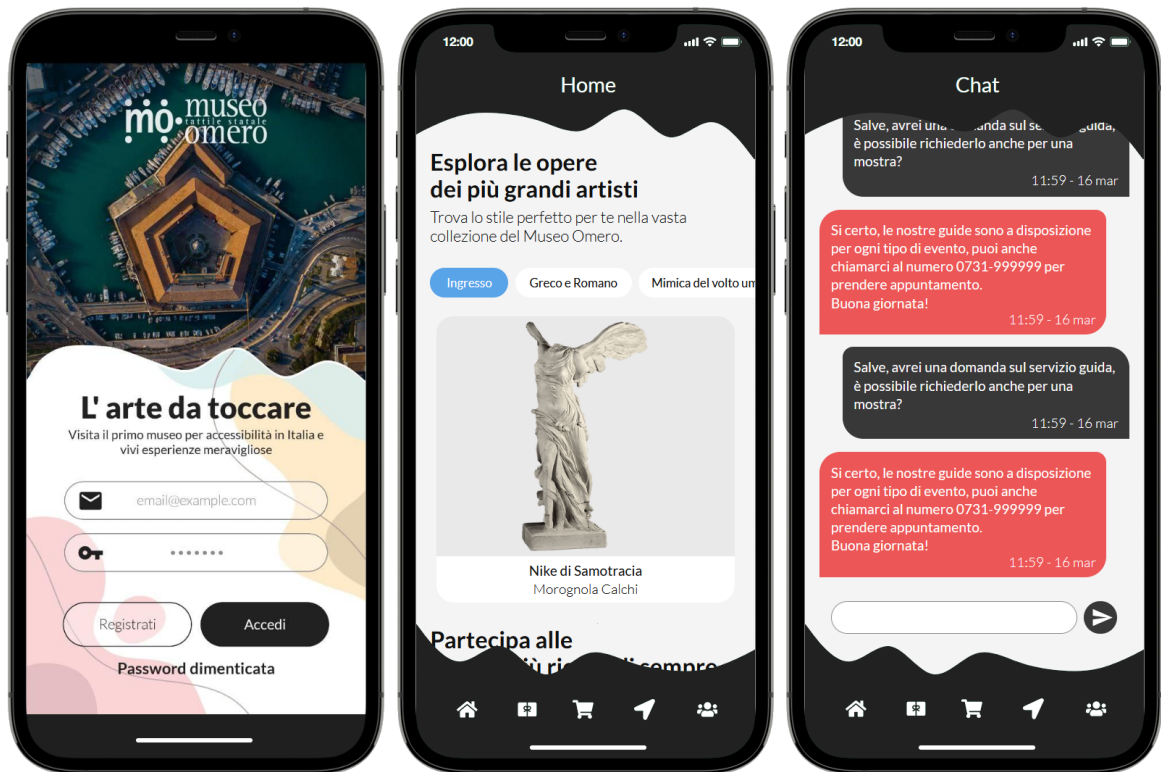


Figure 4.10: The mockup for the customer application’s login, home and chat pages

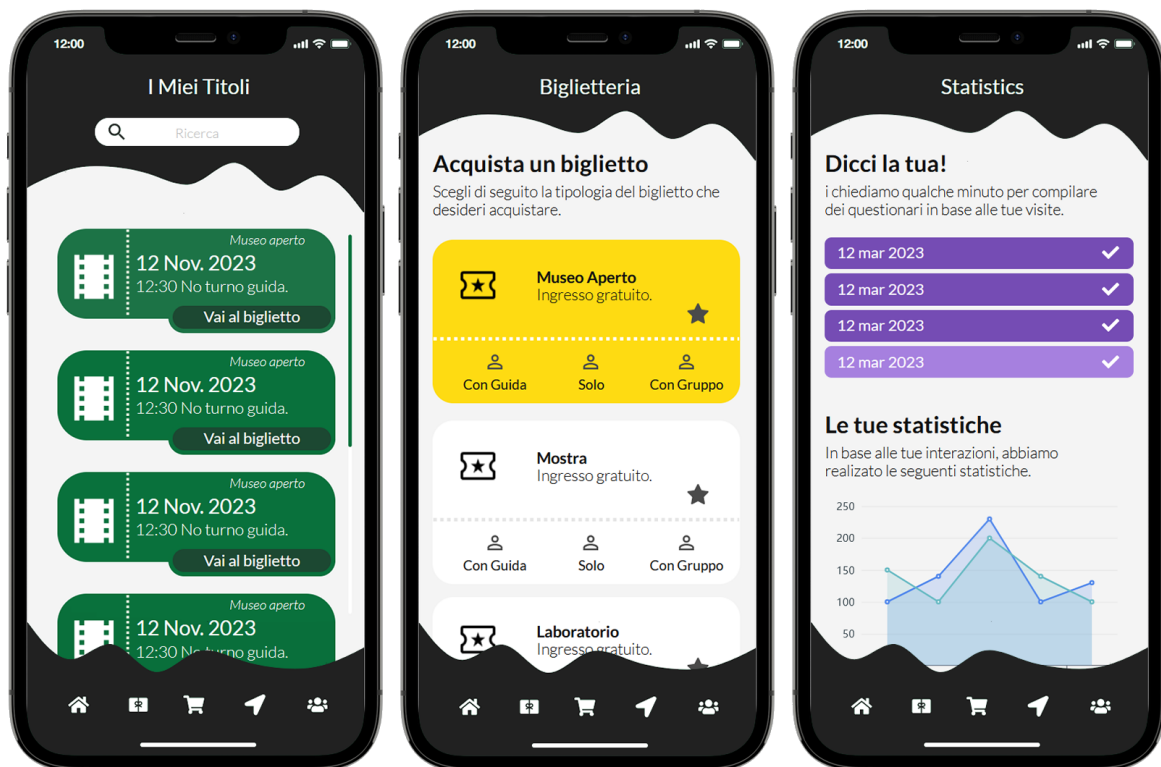


Figure 4.11: The mockup for the customer application’s tickets, ticket office and statistics pages

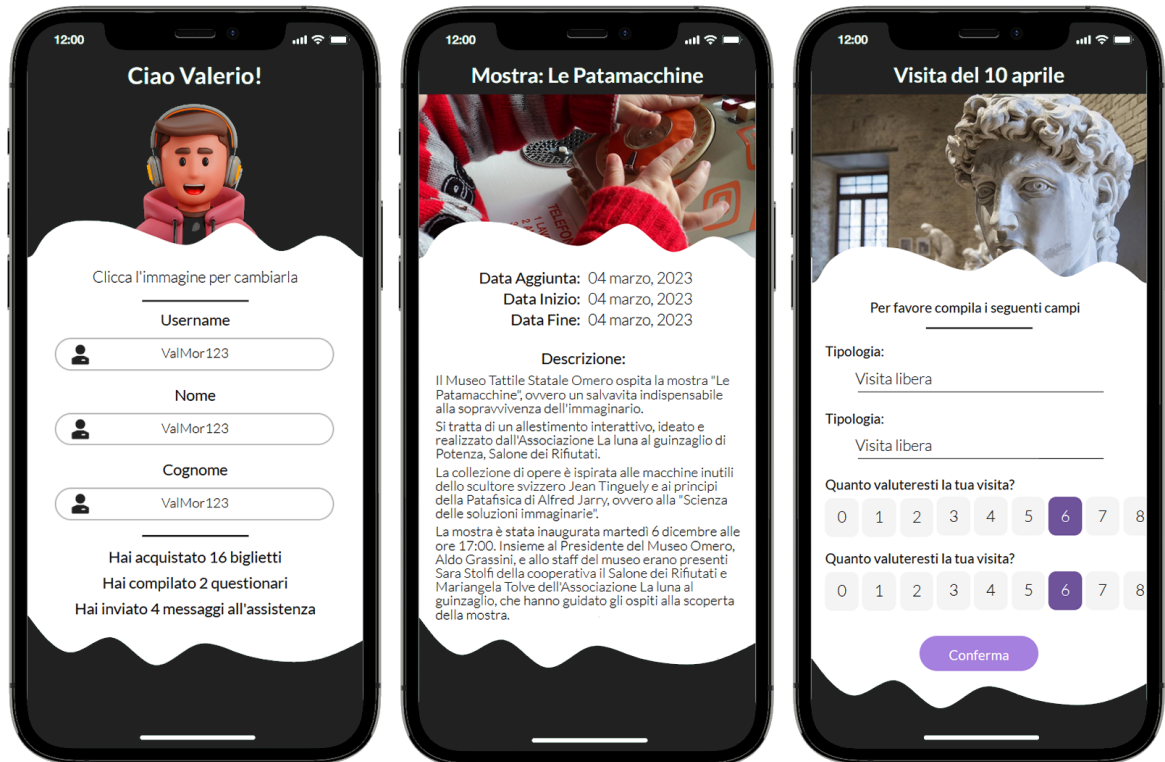


Figure 4.12: The mockup for the customer application's account, exhibit and questionnaire pages

4.3.2 Overall software structure

With a well-defined architectural pattern in mind, the software's structure has been designed to align with it. Consequently, the three namespaces "Model," "View," and "View-Model" are expected to be present. Furthermore, the decision to incorporate the "Managers" namespace, which is shown in Figure 4.13 has also been reached. Within it, a series of singleton classes that execute essential low-level operations pivotal to the software's functionality will be meticulously developed. They include interactions with the remote database and user credential management services, among others.

Figure 4.14 reports the refined model classes diagram, contextualised to implementation aspects. Foremost among these is the usage of the *Newtonsoft.Json* library, employed for the automatic serialization and deserialization of objects into maps. This integration facilitates seamless communication with the remote database.

In particular, through the usage of *JsonProperty* and *JsonIgnore* stereotypes, it is possible to instruct the library to include or exclude specific fields during the serialization of instances in the database. Moreover, the *JsonConstructor* stereotype is adopted to disambiguate cases where multiple constructors are present, ensuring clarity in the deserialization process.

In accordance with the explanation provided in the previous section, a reasonable choice has been made to categorize the view classes under the namespaces "ViewStaff" and "View-Customer," while the view model classes will belong to the namespaces "ViewModelStaff" and "ViewModelCustomer". The former arrangement is shown in Figure 4.16, while the latter is depicted in Figure 4.15.

The design of such classes will naturally take into account the functionality provided by the MAUI framework on which the software will rely. Hence, it should not surprise that methods suffixed with "_Clicked," "_Tapped," and similar terminologies will manifest inside the view classes, in accordance with the event-driven programming paradigm adhered to by

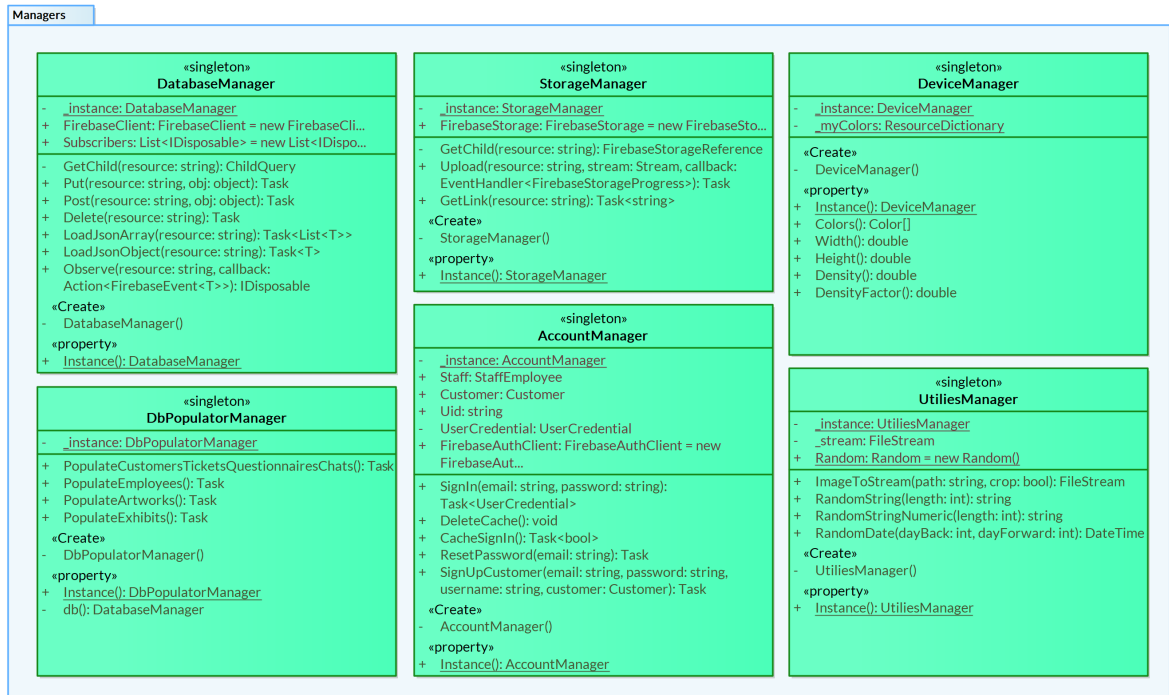


Figure 4.13: The design class diagram for the managers classes

the .NET rendering engine.

4.4 Sequence diagrams

UML sequence diagrams serve as powerful visual representations used to illustrate the dynamic interplay among various software components. These diagrams plainly depict the dynamic interactions and chronological message flow between those components during the execution of a specific scenario. In this way, they serve as a valuable tool for portraying complex communication dynamics at an early stage of the design process. The underlying philosophy aligns with the object-oriented programming principle, allowing the segmentation of interacting parts into objects.

However, it is important to note that these diagrams, much like the others made during this design phase, are targeted towards developers who will assume the responsibility of code implementation in the subsequent development phase. As a result, the description of implementation-related aspects holds noteworthy significance within this context.

A detailed illustration of each use case discussed in Section 3.2.2 will not be provided here. Instead, only those scenarios in which complex communications occur between the software and external data and credential management services will be considered. In detail, the considered use cases are *Login (UC 01)*, *ResetPassword (UC 02)*, *UploadAvatarImage (UC 15)* and *SignUp (UC 17)* and are depicted in Figures 4.17 - 4.20.

To improve readability, the diagrams are accompanied by a natural language description on the side. Moreover, the entity-control-boundary pattern was employed. This architectural pattern finds application in use-case-driven object-oriented software design, organizing the arrangement of classes according to their roles within the system. Specifically, the classes engaged in the delineated scenarios are systematically classified into three distinct sets:

- *Boundaries*, including elements responsible for managing interactions with external actors, including views and remote services,

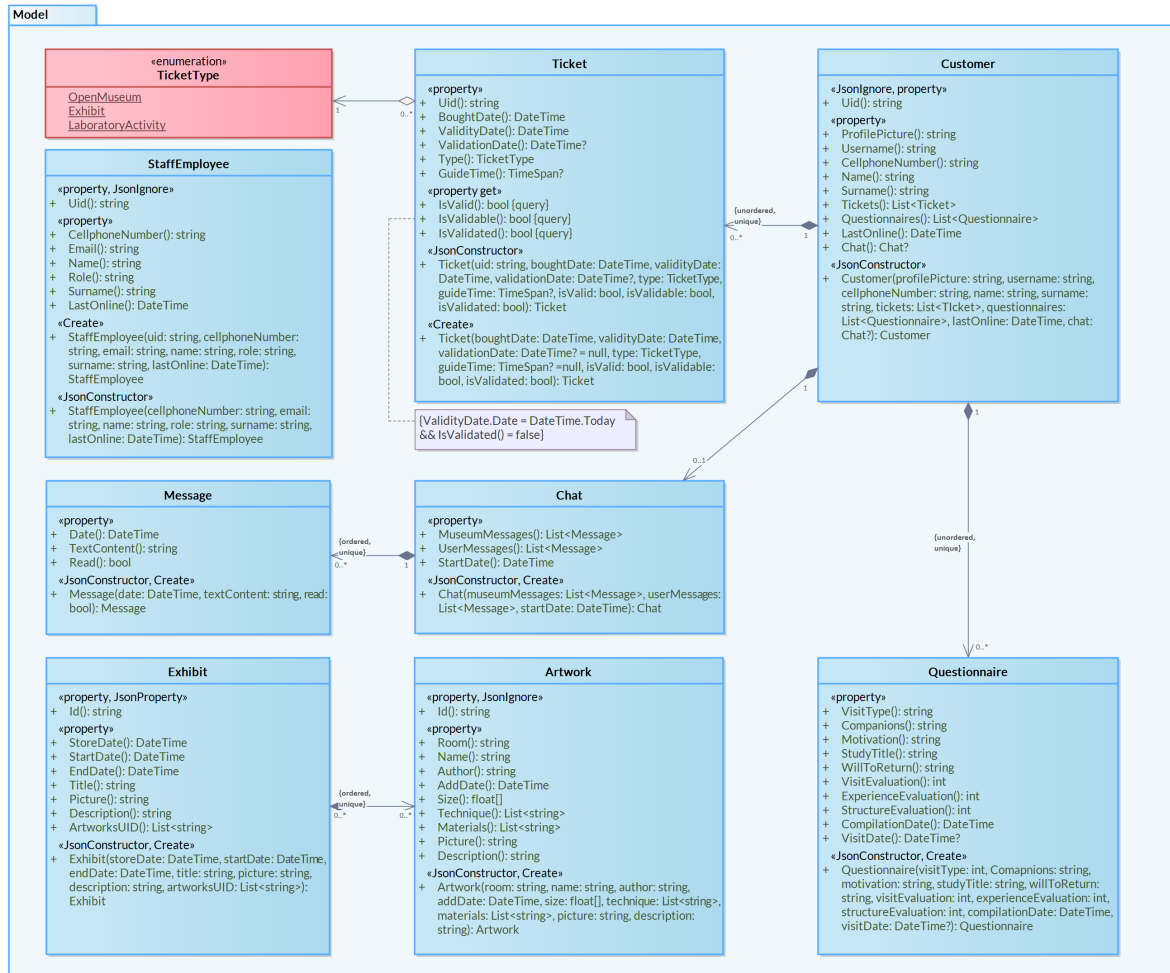


Figure 4.14: The design class diagram for the model’s classes

- *Controls*, ensuring the necessary processing for executing use cases and their associated business logic,
- *Entities*, containing information of significance to stakeholders, typically holding a state of persistence.

4.5 Activity diagrams

Activity diagrams are UML behaviour diagrams that serve a very similar purpose as the sequence diagrams seen in Section 4.4. They provide a structured visual representation of sequence workflows in the system. These diagrams offer a high-level view of how activities interconnect, highlighting the sequence, conditions, and concurrency of tasks. They aid in defining complex logic activities and supporting the codification process of such algorithms.

Similarly to the selection process for sequence diagrams, only use cases demonstrating nontrivial algorithmic complexity were chosen in this context. In particular, the considered use cases are *ExhibitsCUD (UC 06)*, *ValidateTicket (UC 08)*, and *SendMessage (UC 14)* and are depicted in Figures 4.17 - 4.20.

It is worth noting that the *datastore* stereotype was employed to represent the remote database holding the instances; the communication with it is mediated by the *send signal* and *receive signal* elements, depicted by red arrows.

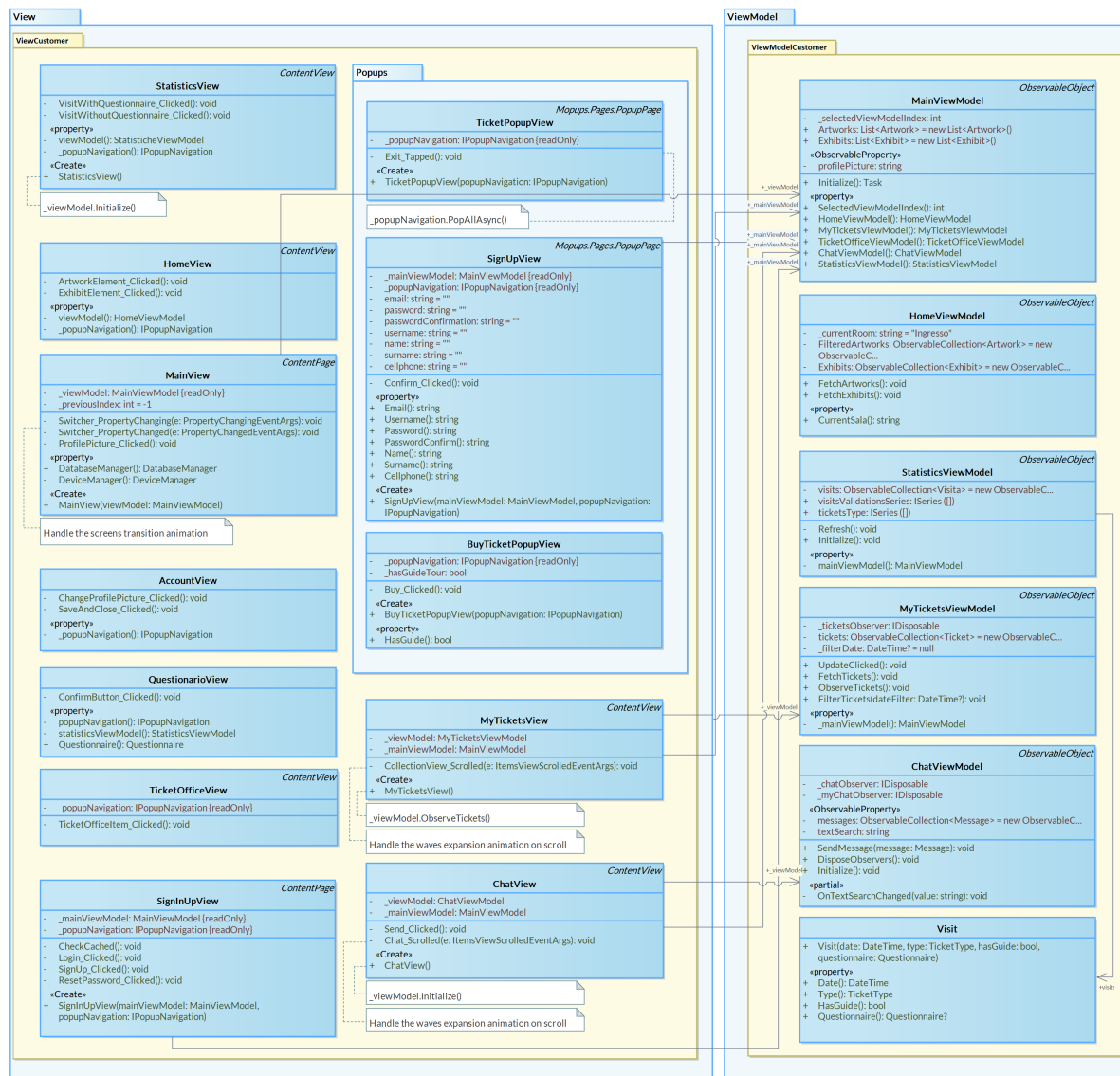


Figure 4.15: The design class diagram for the view and view model classes of the customer application

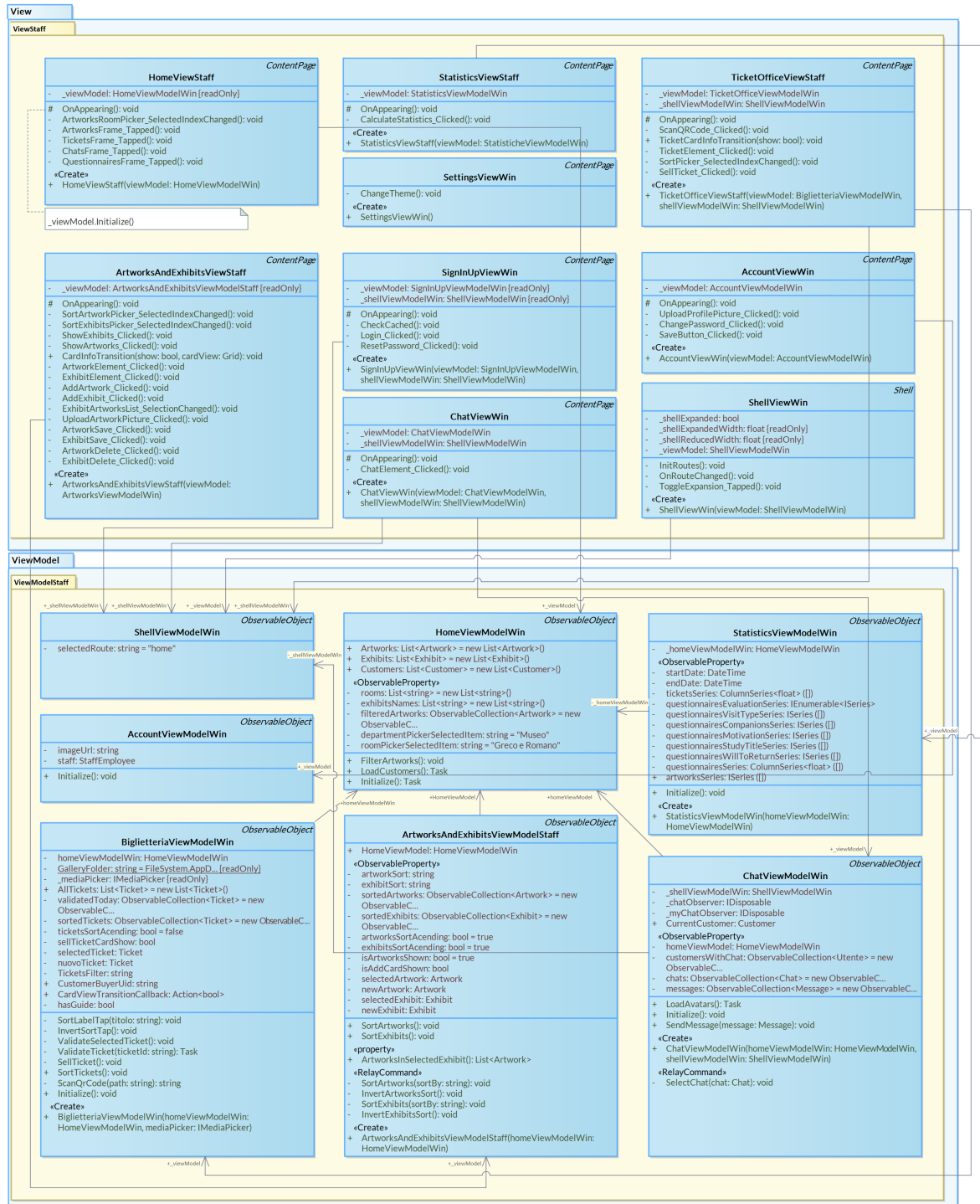


Figure 4.16: The design class diagram for the view and view model classes of the staff application

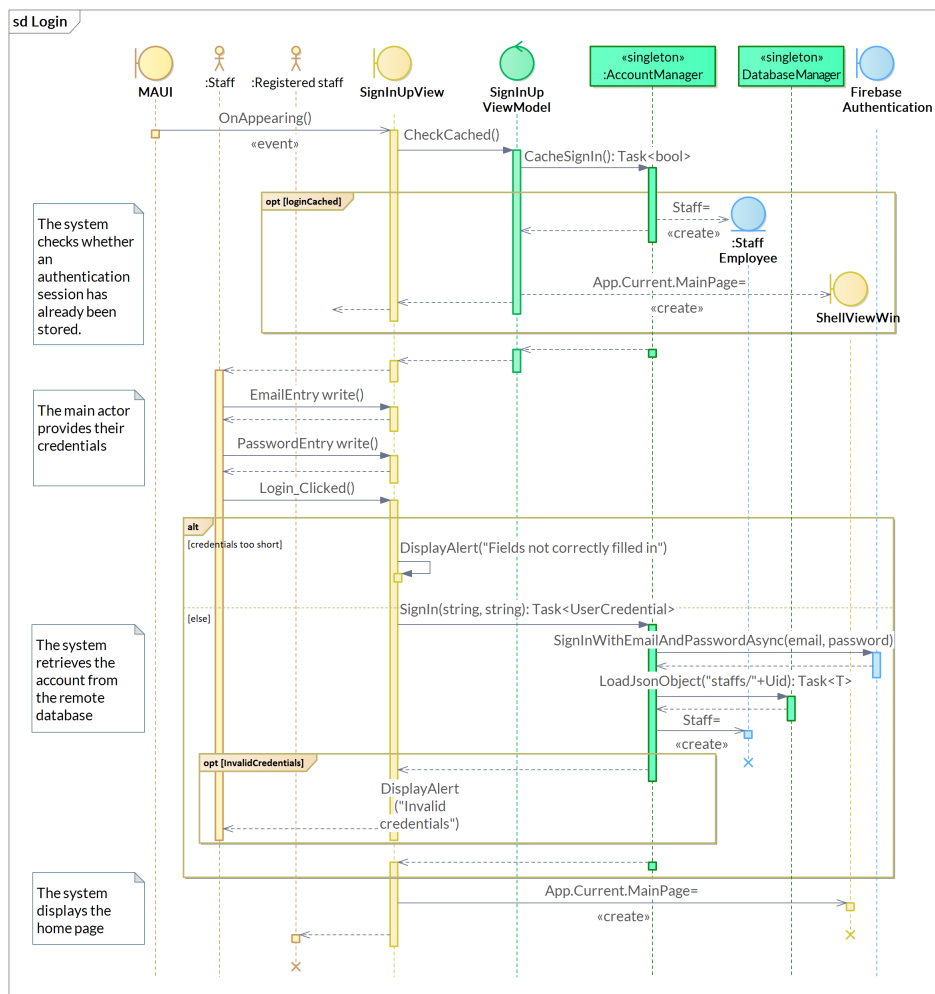


Figure 4.17: The sequence diagram for the *Login* use case

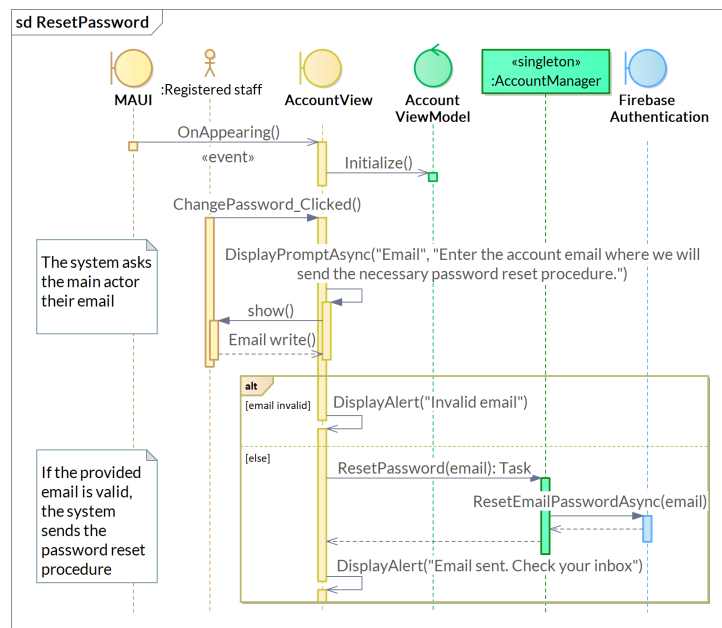


Figure 4.18: The sequence diagram for the *ResetPassword* use case

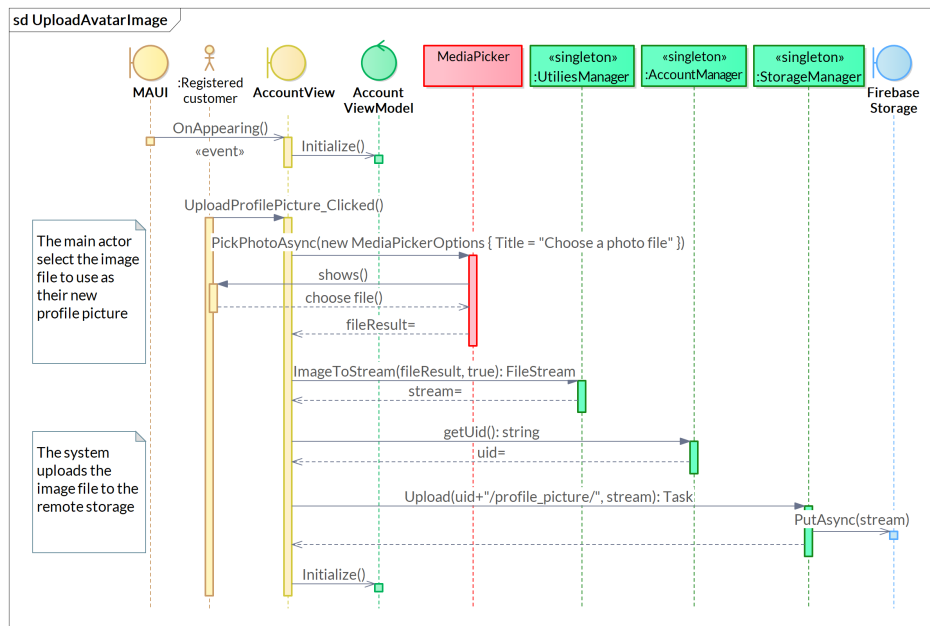


Figure 4.19: The sequence diagram for the *UploadAvatarImage* use case

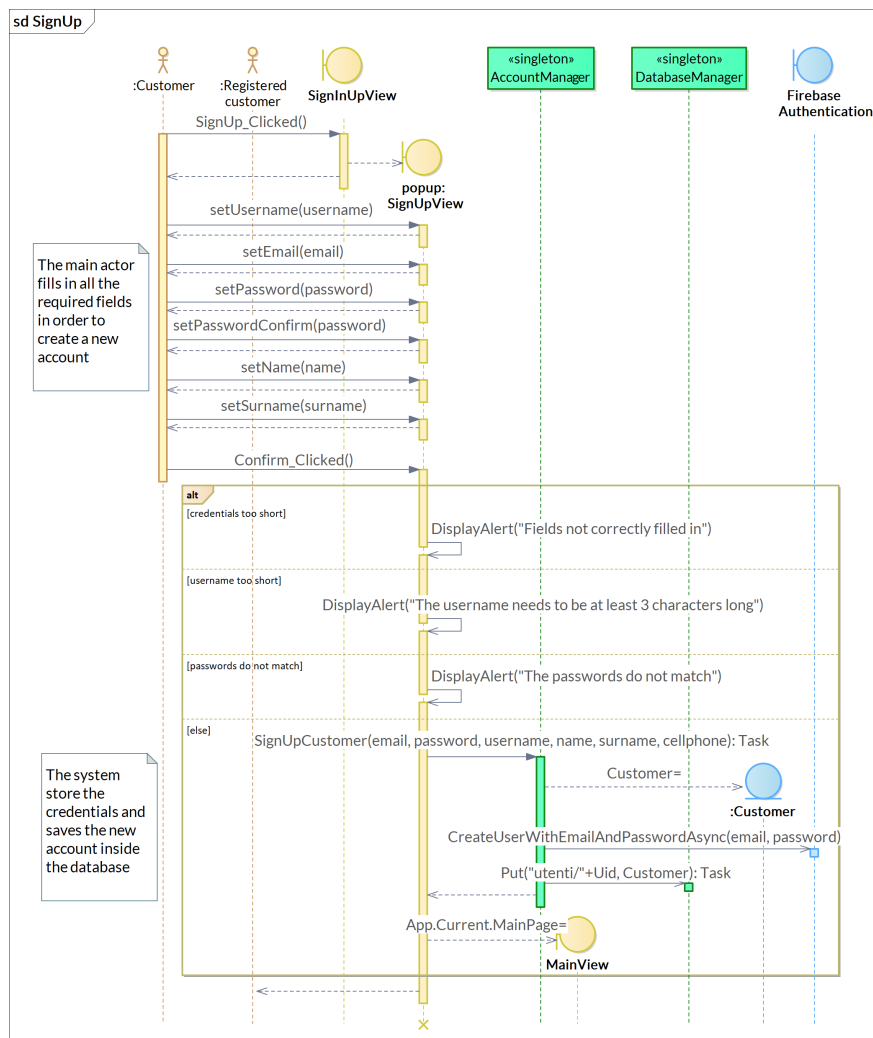


Figure 4.20: The sequence diagram for the *SignUp* use case

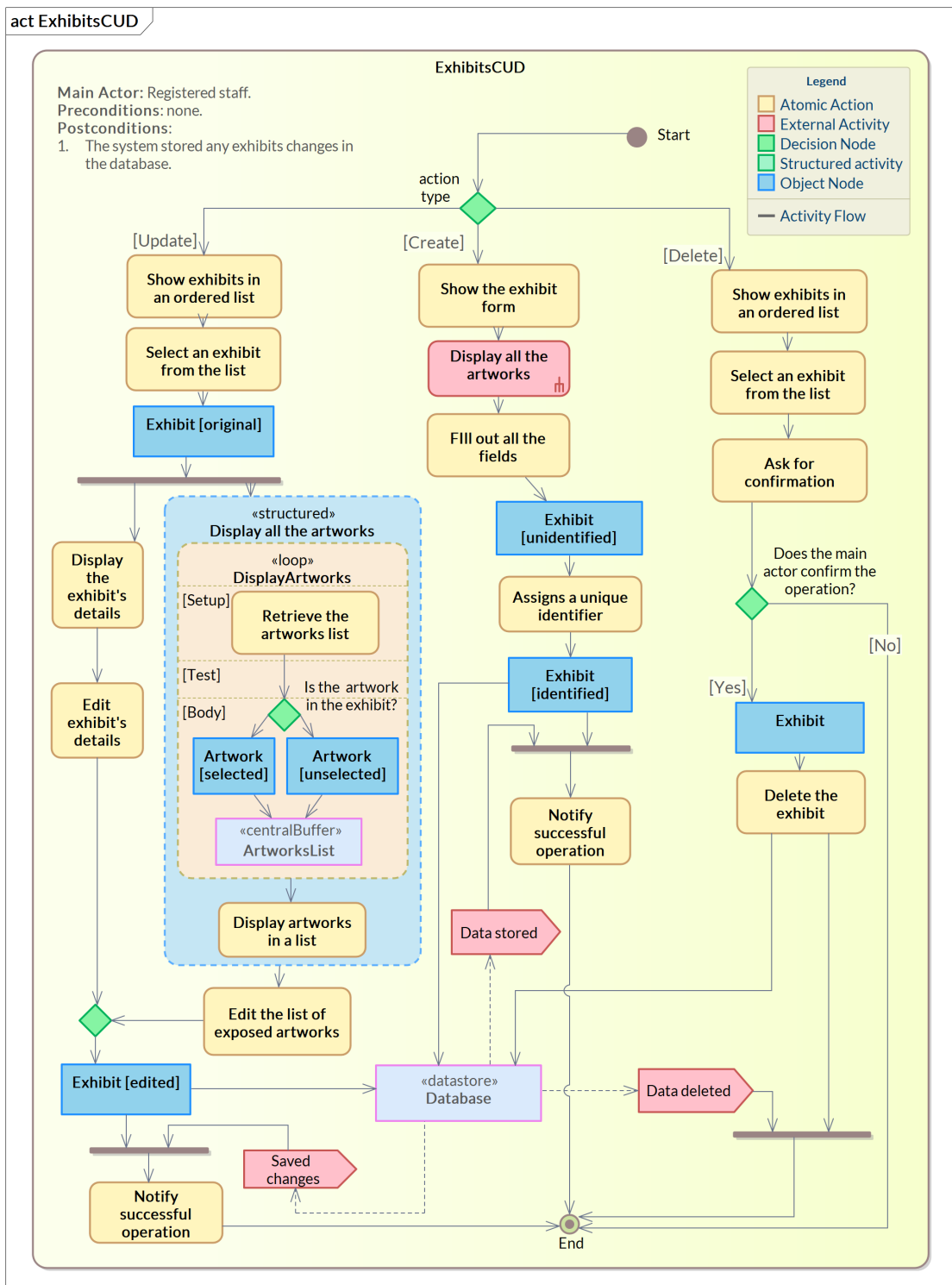


Figure 4.21: The activity diagram for the ExhibitsCUD use case

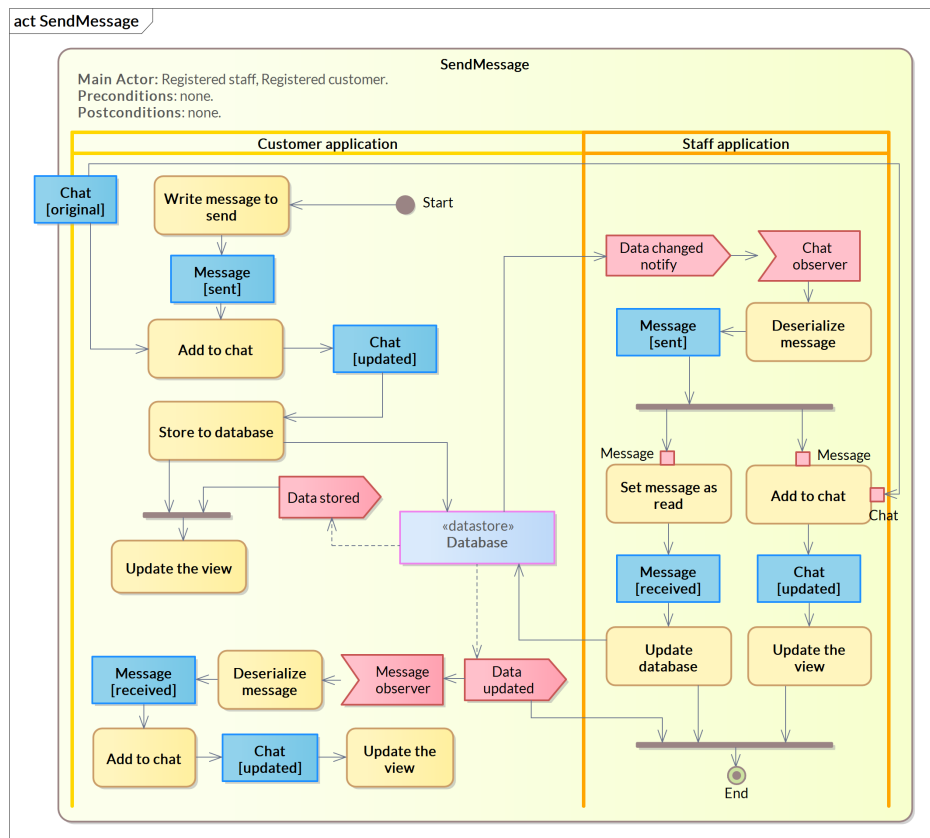


Figure 4.22: The activity diagram for the *SendMessage* use case

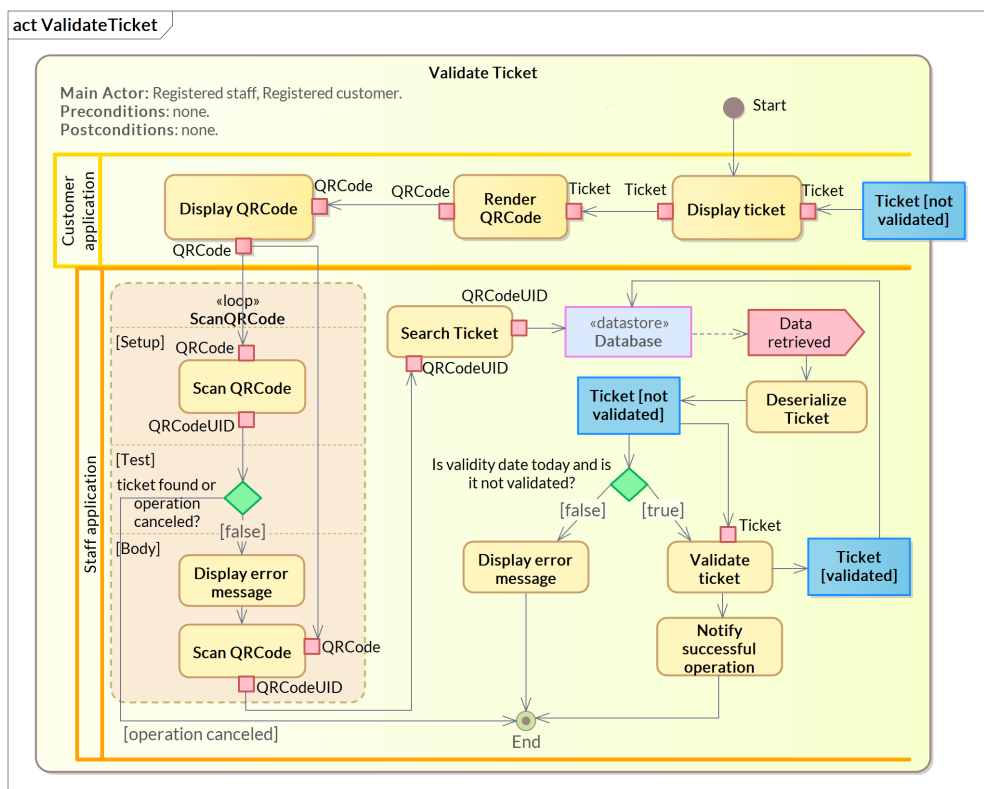


Figure 4.23: The activity diagram for the *ValidateTicket* use case

With the completion of the software design, the focus now shifts to the coding phase. As delineated in Section 5.1, the implementation phase starts with the installation of prerequisite technologies, most notably the MAUI framework, followed by the integration of Firebase services. Subsequently, Section 5.2 elaborates on the aspects of coding the previously defined views. This involves populating global resources and importing requisite NuGet packages. Section 5.3 then discusses the establishment of algorithms for generating random database entries for testing purposes. The chapter concludes in Section 5.4, where we enumerate the primary challenges encountered during the engagement with the MAUI framework.

5.1 Development technologies

The selection of software tools to support system development is directly influenced by the non-functional requirements specified in Section 3.1.2. Specifically, the choice to employ the .NET MAUI framework necessitates the use of platforms developed by Microsoft. As for the *Integrated Development Environment (IDE)*, the selection is constrained to Visual Studio; however, the free Community version suffices for the project's requirements. It's worth noting that, at the time of this software's implementation, Visual Studio Code was not fully compatible with the MAUI framework, lacking several key features, such as hot reload.

Visual Studio serves as an invaluable resource for developers, providing extensive support features. One of its notable integrations is GIT, a widely utilized *Version Control System (VCS)*. This integration streamlines the project management process considerably. With just a few clicks, developers can create a GIT repository for the project, allowing for regular commits throughout the development cycle. This capability not only facilitates tracking the software's evolution but also enables seamless rollback operations in the event of issues. Moreover, it is possible to configure the personal *GitHub* account in order to push the commits to a dedicated remote repository.

Given that the MAUI framework is cross-platform, the IDE offers compatibility with Google's Android emulation system, as well as the option to connect to local or remote iOS devices. In this project, due to the absence of both a physical iOS device and a Mac computer, development has been primarily focused on Windows and Android platforms. However, this choice does not undermine the cross-platform capabilities of the project granted by the framework.

To run the application on an Android platform, an *Android Virtual Device (AVD)* was

chosen over a physical Android device due to its superior flexibility and execution speed. Following Google’s documentation, the task of creating a new virtual device using the Android Device Manager proved to be straightforward. The only requirements were to specify the amount of hardware resources to virtualize and to select the version of the operating system’s API for installation.

5.1.1 The MAUI framework

As indicated by Microsoft’s documentation, the installation process for the components required to work with the MAUI framework is considerably more straightforward compared to its predecessor, Xamarin. The intricacies of the installation process are abstracted away from the user, requiring only a few selections within the Visual Studio Installer graphical interface, as depicted in Figure 5.1.

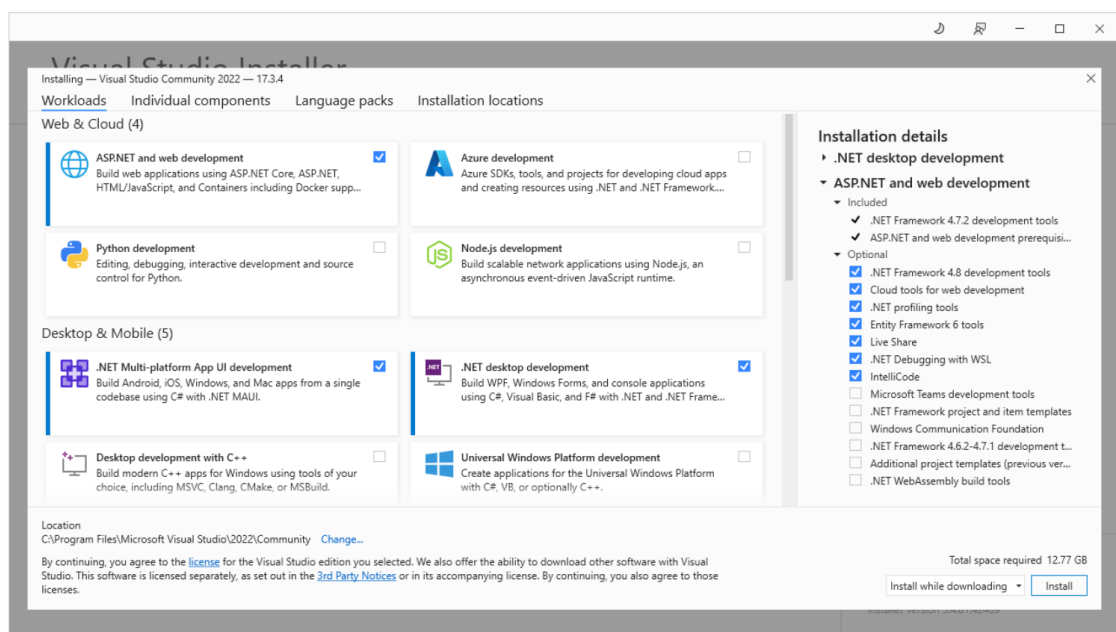


Figure 5.1: Installation of MAUI framework in Visual Studio Installer 2022

Once the required components are installed, the next task is to set up a new MAUI project. Using the IDE’s intuitive interface, a new folder with a basic MAUI app structure is created. Next, the focus shifts to outlining how the software will work, especially its startup behaviour. Since the software is expected to show different screens based on the device it’s used on, changes to the MAUI starting point are needed.

For this purpose, the `App.xaml.cs` file required editing. This class serves as the backbone of the whole application, designating shared resources and identifying the initial view to be displayed. As depicted in Figure 5.2, a precompiler directive was utilized to conditionally select the view to be loaded.

In the event that the application to be run is the staff application, the initial view to be loaded is `SignInUpViewWin`. The latter subsequently transitions to `ShellViewWin` upon successful login or if login credentials are cached. Conversely, if the application in operation is the customer application, the initial view to be displayed is `SignInUpView`, which will then transition to `MainView` following a successful login.


```

public partial class App : Application
{
    #if WINDOWS || MACCATALYST
        1 riferimento
        public App(SignInUpViewModelWin signInUpViewModelWin, ShellViewModelWin shellViewModelWin)...
        3 riferimenti
        private void LoadTheme(string theme)...
    #elif ANDROID || IOS
        public App(MainViewModel mainViewModel, IPopupNavigation popupNavigation)...
    #endif
}

```

Figure 5.2: The `App.xaml.cs` file representing the software entry point

5.1.2 Firebase services

As required by the non-functional requirements detailed in Section 3.1.2, the software will leverage various Firebase services to implement key functionalities, such as credential management, data storage, and database systems. These include:

- *Firebase Authentication;*
- *Firebase Realtime Database;*
- *Firebase Storage.*

Upon establishing a subscription with the platform, a new project was created and the Spark pricing plan was selected. This free-tier plan allows developers access to a broad range of services within reasonable usage limits. In the current project context, these constraints are more than adequate as the application is not expected to exceed a 1GB database quota or maintain more than 100 simultaneous connections. Moreover, the flexibility of Firebase makes transitioning to a paid plan straightforward, requiring only a few clicks.

To enable the software to interact with these Firebase services, suitable *NuGet* packages were installed from the marketplace to streamline communication with the *RESTful* API. As depicted in Figure 5.3, these consist of three unofficial libraries that facilitate functionalities like cache management and handle *WebSocket* connections, thereby allowing for real-time monitoring of changes in the *Realtime Database*

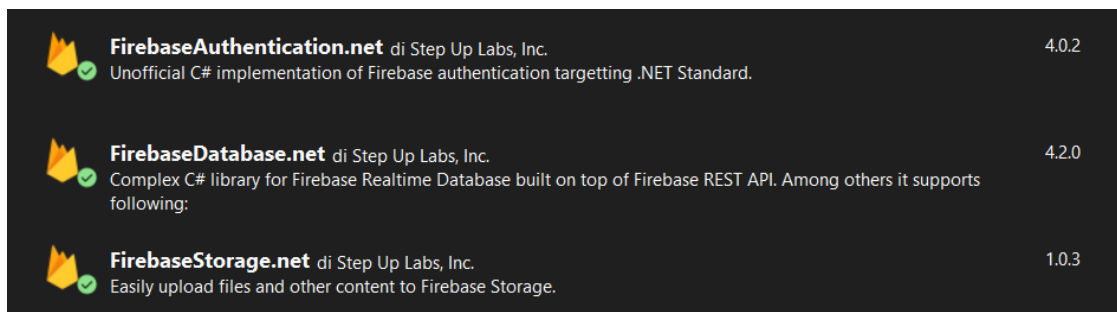


Figure 5.3: The three *NuGet* packages that integrate the corresponding Firebase services

The implementation phase proceeds with the population of the `Manager` namespace, composed of the `AccountManager`, `DatabaseManager`, and `StorageManager` singleton classes. As previously specified, these classes act as abstractions for the low-level operations crucial for executing the business logic in the view model classes. These methods

are consistent with those delineated in the refined class diagram seen in Section 4.3 and interact directly with the APIs provided by the aforementioned libraries.

A noteworthy aspect involves the methods responsible for deserializing objects and lists of objects from the remote database. Given the complexity often associated with this operation across platforms, the `Newtonsoft.Json` library was chosen to facilitate object-to-map translation, as shown in Figures 5.4 and 5.5. Here, through the application of generics, the client code specifies the class to be used as the target for deserialization when invoking the method for database object retrieval. Depending on whether a single instance or a list is being retrieved, the static `DeserializeObject` method of the `JsonConvert` class is called, applying the relevant generic type parameters. For lists of objects, the generics adopt the type `Dictionary<string, T>` as it is certain that the key for each instance corresponds to its Unique Identifier (UID).

```
public async Task<T> LoadJsonObject<T>(string resource)
{
    var child = GetChild(resource);
    var collection = await child.OnceAsJsonAsync();
    var obj = JsonConvert.DeserializeObject<T>(collection);
    if (obj is { })
    {
        if (typeof(T) == typeof(Staff))
            (obj as Staff).Uid = resource.Split('/',
                StringSplitOptions.RemoveEmptyEntries).Last();
        if (typeof(T) == typeof(Customer))
            (obj as Customer).Uid = resource.Split('/',
                StringSplitOptions.RemoveEmptyEntries).Last();
    }
    return obj;
}
```

Figure 5.4: The `LoadJsonObject` method of the `DatabaseManager` class used to retrieve an instance from the remote database

```
public async Task<List<T>> LoadJsonArray<T>(string resource)
{
    var collection = await GetChild(resource).OnceAsJsonAsync();
    var dict = JsonConvert.DeserializeObject<Dictionary<string, T>>(collection);
    if (dict is null)
        return null;
    if (typeof(T) == typeof(Customer))
        foreach (var entry in dict)
            (entry.Value as Customer).Uid = entry.Key;
    if (typeof(T) == typeof(Artwork))
        foreach (var entry in dict)
            (entry.Value as Artwork).Id = entry.Key;
    if (typeof(T) == typeof(Exhibit))
        foreach (var entry in dict)
            (entry.Value as Exhibit).Id = entry.Key;
    return (from entry in dict select entry.Value).ToList();
}
```

Figure 5.5: The `LoadJsonArray` method of the `DatabaseManager` singleton class used to retrieve a list of instances under a specified node on the remote database

It is important to note that, in both scenarios, the UID variables within the instances are populated using the keys of the object nodes themselves. This approach ensures the preservation of information that would otherwise be lost, as the deserialization process is confined solely to the node's value. It avoids the need for serializing the UID field when storing the

instance back into the database too, thereby eliminating unnecessary redundancy. To achieve this, the `JsonIgnore` annotation, provided by the `Newtonsoft.Json` library, was employed, as previously detailed in Section 4.3.2.

With regards to database access security, simple rules were implemented, as illustrated in Figure 5.6. These rules allow CRUD operations only to authenticated users, resulting in a fairly permissive security model.

```
// Only authenticated users can access/write data
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

Figure 5.6: The rules managing the access to the *Realtime Database*

On the coding front, as illustrated in Figure 5.7, it is possible to specify the active user session by passing an instance of `FirebaseOptions` during the instantiation of the `FirebaseClient` object, which outlines the procedures for acquiring the user's authentication token. This object serves as an abstraction layer for communications with the *Realtime Database* service. The required token is obtained by interrogating the *Authentication* service, which is presumed to have a validated session in place at the time of database querying.

```
public FirebaseClient FirebaseClient =
    new(
        "https://museoomero-ca8aa-default-rtdb.europe-west1.firebaseio.com/",
        new FirebaseOptions
        {
            AuthTokenAsyncFactory = () =>
                AccountManager.Instance.FirebaseAuthClient.User.GetIdTokenAsync()
        });
```

Figure 5.7: The creation of the `FirebaseClient` object with the user token for recognition

With respect to the methods within the `AccountManager` and `StorageManager` classes, which pertain to the *Authentication* and *Storage* services respectively, a comprehensive description is deemed unnecessary. This is because the code of their methods, essentially involves straightforward invocations of functions from previously imported libraries, presenting no significant complexities.

5.2 Views implementation

Upon completing the implementation of foundational classes that interface with Firebase services, the focus shifts toward the development of components adhering to the MVVM architectural pattern. It is worth noting that while the creation of model classes has not been explicitly discussed, these were directly extracted from the class diagrams generated during the design phase in Section 4.3. In fact, the encoding of these classes into `.cs` files was made possible by defining the target programming language within the adopted UML diagramming tool. Then, these files were readily integrated into the solution with only some minimal adjustments needed.

In contrast, developing views and view model classes presented non-trivial complexities, necessitating careful coding to integrate with the MAUI framework. For this purpose, in the

next sections, we will delve into some of these development aspects and explain the strategies employed to resolve them.

To enhance organizational clarity, these classes have been grouped into four packages: `ViewStaff` and `ViewModelStaff` for the staff application, and `ViewCustomer` and `ViewModelCustomer` for the customer application.

5.2.1 Supporting components

Early in the views development process, it became evident that establishing a standardized visual style for the creation of views was crucial. This entailed the selection of consistent components like an icon set, a choice of fonts, and a color palette that would be uniformly applied across various screens of the application. The following will provide a brief overview of each of these key design elements.

As already announced, one significant aspect emerging from the mockups exposed in Section 4.2.2 is the need for an icon pack. To address this requirement, we opted to use Google’s Material Icons.

The advancements introduced by the MAUI development team have dramatically improved resource management, offering distinct advantages over its predecessor, Xamarin. The official documentation strongly recommends using vector images, when it comes to handling picture assets, allowing the framework to dynamically rasterize these into *PNG* format based on the device’s screen resolution [Britch, 2023a]. This recommendation aligns well with the Google Material Icons, which can be conveniently exported in *SVG* vector format directly from their website.

Nevertheless, an alternative approach was chosen to enhance the flexibility in selecting icons. Rather than incorporating individual *SVG* files, the complete Material Icons set was imported as a singular font file. This file was subsequently stored in the designated `Resources/Fonts` directory. Then, to streamline its use, each icon was mapped to a static string constant, which was named to reflect the corresponding icon. This approach removes the need to search for the *UNICODE* value each time an icon is required in an *XAML* layout. For this extensive mapping task, Andrei Nitescu’s GitHub repository *IconFont2Code* was utilized which produced the outcome shown in Figure 5.8.

```
namespace MuseoOmero.Resources.Material;
18 riferimenti
static class IconFont
{
    public const string VectorSquare = "\U000f0001";
    public const string AccessPointNetwork = "\U000f0002";
    public const string AccessPoint = "\U000f0003";
    public const string Account = "\U000f0004";
    public const string AccountAlert = "\U000f0005";
    public const string AccountBox = "\U000f0006";
```

Figure 5.8: The `IconFont` class containing a list of all the Material symbols

The class’s static nature is essential for easy access to its fields from all the *XAML* view files. In fact, by defining the *material* namespace with the *xmlns* *XAML*’s global attribute, its fields can be accessed through binding expressions, as detailed in Section 1.5.

After successfully importing the icon pack, the next step involves configuring the fonts for all text elements within the applications. As previously determined during the design phase, the chosen font is Google’s *Lato*.

After downloading the font from its official webpage and importing it into the `Resources` folder, registration within the MAUI application builder is required. This is accom-

plished within the MAUI `CreateMauiApp` static method, which serves as the framework's entry point, as depicted in Figure 5.9.

```
public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();
    builder
        .UseMauiApp<App>()
        .ConfigureFonts(fonts =>
        {
            fonts.AddFont("Lato-Regular.ttf", "Lato");
            fonts.AddFont("Lato-Italic.ttf", "LatoItalic");
            fonts.AddFont("Lato-Light.ttf", "LatoLight");
            fonts.AddFont("Lato-LightItalic.ttf", "LatoLightItalic");
            fonts.AddFont("Lato-Bold.ttf", "LatoBold");
            fonts.AddFont("Lato-Black.ttf", "LatoBlack");
        });
}
```

Figure 5.9: The registration of used fonts

The completion of theme-switching functionality in both applications marks the end of this section. As previously outlined in Section 3.1.2, requirement *NFR-05 Dark theme support* mandates the incorporation of multiple themes in both the staff and customer applications.

For the latter, implementing this feature is relatively straightforward. Mobile operating systems natively offer built-in support for toggling between light and dark themes. The MAUI framework simplifies this process further by offering the `AppThemeBinding` markup extension. This extension allows for conditional object assignment based on the current system theme. By utilizing this extension in conjunction with binding expressions, it is possible to define color schemes that dynamically adapt to the currently active theme within the views.

In the staff application, in accordance with the above non-functional requirement, the capability to switch between different color themes has been introduced. Individual *ResourceDictionary* instances have been created for each available theme, as depicted in Figure 5.10.

<u>GreenTheme.xaml</u>	<u>DarkGreenTheme.xaml</u>
<Color x:Key="Color1">■#34265C</Color>	<Color x:Key="Color1">■#1c4832</Color>
<Color x:Key="Color2">■#137040</Color>	<Color x:Key="Color2">■#137040</Color>
<Color x:Key="Color3">■#FFF79C</Color>	<Color x:Key="Color3">■#b5e8be</Color>
<Color x:Key="Color4">■#f3f3f3</Color>	<Color x:Key="Color4">■#b5e8be</Color>
<Color x:Key="Color5">■White</Color>	<Color x:Key="Color5">■White</Color>
<Color x:Key="Color6">■#212121</Color>	<Color x:Key="Color6">■#8c8c8c</Color>
<Color x:Key="Color7">■#383838</Color>	<Color x:Key="Color7">■#212121</Color>
<Color x:Key="Color8">■#40ffffff</Color>	<Color x:Key="Color8">■#40ffffff</Color>
<u>BlackTheme.xaml</u>	<u>HighContrastTheme.xaml</u>
<Color x:Key="Color1">■#212121</Color>	<Color x:Key="Color1">■#000000</Color>
<Color x:Key="Color2">■#137040</Color>	<Color x:Key="Color2">■#d0b132</Color>
<Color x:Key="Color3">■#b5e8be</Color>	<Color x:Key="Color3">■#f6d13b</Color>
<Color x:Key="Color4">■#f3f3f3</Color>	<Color x:Key="Color4">■#f3f3f3</Color>
<Color x:Key="Color5">■White</Color>	<Color x:Key="Color5">■White</Color>
<Color x:Key="Color6">■#8c8c8c</Color>	<Color x:Key="Color6">■#8c8c8c</Color>
<Color x:Key="Color7">■#1c4832</Color>	<Color x:Key="Color7">■#f6d13b</Color>
<Color x:Key="Color8">■#40ffffff</Color>	<Color x:Key="Color8">■#40ffffff</Color>

Figure 5.10: The different themes for the staff application

Only the *ResourceDictionary* corresponding to the currently selected theme is incorporated into the *MergedDictionaries* collection, namely the list of all the active resources. This operation

is implemented by the `LoadTheme()` method within MAUI's `App` class. Essentially, this method purges the `MergedDictionaries` collection of all existing theme dictionaries before adding the one associated with the current selection.

To enable real-time theme changes without requiring an application restart, the `WeakReferenceMessenger` class from the `CommunityToolkit.MVVM` library was leveraged. The latter allows the `LoadTheme()` method to be exposed, while maintaining a loosely coupled architecture between the `App` and the client classes.

Lastly, it is worth noting that when colors are specified within the XAML views using binding expressions, they are referred to as *DynamicResources* rather than *StaticResources*. This distinction informs the framework to monitor changes to the `MergedDictionaries` and refresh the views accordingly.

5.2.2 Applications' Shells

In the realm of modern application development, the Single-Page Application (SPA) philosophy is increasingly gaining traction [Mendes, 2023]. While primarily associated with web development, this approach is part of a broader agile methodology that emphasizes ease of development over more complex, robust system architectures. Best practices in software engineering advocate for a modular design composed of small, reusable components. The reasoning behind this guideline lies in the principle of encapsulation. By adhering to it, one can focus on developing components that encapsulate specific functionalities.

In the context of the current project, as illustrated in Figures 4.2 and 4.3, the design strategy leans heavily towards using component-based views, specifically subclasses of `ContentView`, rather than relying on container-based views. In essence, both the staff and client applications screens are predominantly housed within a single shell, with the exception of a few standalone pages, such as those for login and registration.

In the design phase, it was already established that the two shells, `ShellViewWin` for the staff application and `MainView` for the client application, would have distinct structural features. Specifically, the former was designed to have a sidebar menu with icons representing each navigable route, while the latter appears as a bottom navigation bar. However, both serve as the primary navigation components for most screens in both applications, becoming accessible to the user only after successful authentication via the initial login view. The implementation of these login screens is not detailed here as they simply utilize the methods from manager classes previously discussed in Section 5.1.2.

Though the default appearance of MAUI's shell component falls short of meeting the specific design objectives, the framework is engineered for flexibility. Specifically, the `Shell` class includes a `FlyoutContentTemplate` attribute of the `DataTemplate` type, allowing for extensive customization of individual menu elements. Figure 5.11 illustrates the implementation of this template for the first menu item.

In line with the design mock-ups, the currently active route should stand out from the rest. Specifically, both the icon and the text should be displayed in white rather than grey colour, a white rectangle should appear on the left, and the icon should be filled rather than outlined. Figure 5.12 clearly depicts these intended visual distinctions.

To meet these dynamic behaviour requirements, there was no necessity to imperatively codify them in the code-behind. As illustrated in Figure 5.11, each menu item is encapsulated within a `RadioButton` element. This serves as the holder for the `Boolean` selection state, thereby enabling conditional assignments to different properties of the menu item. For this task, advanced binding expressions and `VisualStateGroups` were employed, both targeting the same goal. The former utilizes a converter to transform the `boolean` selection state, retrieved from the parent `RadioButton` using the `AncestorType` property, into an appropriate object. The latter, on the other hand, is used to control the visibility of the white rectangle on the menu item.


```

<!--Home-->
<RadioButton Value="home"
  IsChecked="True"
  Grid.Row="1"
  CheckedChanged="OnMenuItemChanged"
  Style="{StaticResource FlyoutItemRadioButton}">
  <RadioButton.Content>
    <Grid Style="{StaticResource FlyoutItemGrid}">
      <Label Style="{StaticResource FlyoutItemIcon}">
        <Label.Text>
          <Binding Source="{RelativeSource AncestorType={x.Type RadioButton}, AncestorLevel=1}"
            Path="IsChecked">
            <Binding.Converter>
              <converters:BoolToObjectConverter
                x.TypeArguments="x:String"
                FalseObject="{Static material:IconFont.HomeVariantOutline}"
                TrueObject="{Static material:IconFont.HomeVariant}" />
            </Binding.Converter>
          </Binding>
        </Label.Text>
      </Label>
      <Label Style="{StaticResource FlyoutItemText}"
        Text="Home" />
    </Grid>
  </RadioButton.Content>
</RadioButton>

```

Figure 5.11: The implementation of the menu item in the staff application

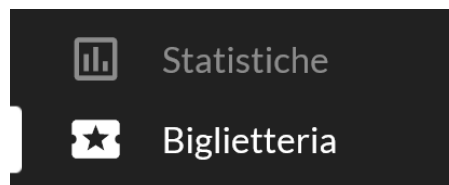


Figure 5.12: The visual behaviour of the flyout menu item

The customer application's bottom bar was implemented using an external component, stemming from the limitations in customizing the appearance and behaviour of MAUI's native `TabbedPage`. The `Sharpnado.Tabs NuGet` package was chosen for its highly customizable layouts and its ability to reduce shell loading times by adopting a lazy loading strategy for individual pages, thereby delaying their runtime creation as much as possible.

The structure of the `MainView` XAML file is essentially divided into two key parts:

- The screen's center hosts an instance of Sharpnado's `ViewSwitcher` class. This component holds the list of all navigable screens, encapsulated within `LazyView` and `DelayedView` instances, depending on their rendering priorities, as illustrated in Figure 5.13.
- Occupying the screen's bottom area is Sharpnado's `TabHostView`, which serves as the bottom bar. This component holds a list of `BottomTabItem` instances, each corresponding to a different screen, as depicted in Figure 5.14.

These two components operate independently, except for a shared `SelectedIndex` property. This property is synchronized through the use of a two-way binding expression, as shown in Figure 5.14.

5.2.3 NuGet packages

During the screen implementation phase for both shells, a variety of `NuGet` packages were utilized to support more complex requirements.

```

<tabs:ViewSwitcher x:Name="Switcher"
    Grid.RowSpan="3"
    Margin="0"
    Animate="True"
    SelectedIndex="{Binding SelectedViewModelIndex, Mode=TwoWay}"
    PropertyChanging="Switcher_PropertyChanging"
    PropertyChanged="Switcher_PropertyChanged">
    <tabs:LazyView x:TypeArguments="views:HomeView"
        AccentColor="{StaticResource Color1}"
        BindingContext="{Binding HomeViewModel}"
        Animate="True"
        UseActivityIndicator="False" />
    <tabs:DelayedView x:TypeArguments="views:MyTicketsView"
        AccentColor="{StaticResource Color1}"
        BindingContext="{Binding MyTicketsViewModel}"
        Animate="True"
        UseActivityIndicator="True" />

```

Figure 5.13: The implementation of the Sharpnado’s `ViewSwitcher` class

```

<tabs:TabHostView Grid.Row="1"
    HeightRequest="60"
    Padding="32,0"
    HorizontalOptions="Center"
    IsSegmented="True"
    CornerRadius="30"
    SelectedIndex="{Binding Source={x:Reference Switcher},
        Path=SelectedIndex, Mode=TwoWay}">
    <tabs:BottomTabItem x:Name="Tab1"
        Style="{StaticResource BottomTab}"
        Label="{x:Static material:IconFont.Home}"
        LabelSize="{Binding FontSize1}" />

```

Figure 5.14: The implementation of the Sharpnado’s `TabHostView` class

Two key packages were `CommunityToolkit.Maui` and `CommunityToolkit.Mvvm`, both of which proved invaluable in streamlining the development process.

The `CommunityToolkit.Maui` package offers a host of useful components that enhance the application’s features. Specifically, several converters like `EnumToIntConverter` for `Picker` elements and `InvertedBoolConverter` were used to make possible complex binding expressions in the application.

On the other side, the `CommunityToolkit.Mvvm` package was employed extensively across all view model classes. This package excels in simplifying the management of bindable properties and methods, thanks to its useful annotations and classes. These aspects have been explained in greater detail in Section 1.4.2.

Regarding ticket validation, the software needs to manage two critical tasks: the on-screen rendering of a QR code-encoded ticket identifier and its decoding via a webcam on the staff application device. Unfortunately, the MAUI framework does not offer built-in support for QR code encoding at the time of this development. Therefore, a third-party solution was needed. The `ZXing NuGet` package, offers a versatile `BarcodeGeneratorView` component that meets the encoding needs, even supporting various encoding types beyond the one in use.

For the decoding aspect, `ZXing` offers a robust library too. As illustrated in Figure 5.15, the software converts the webcam feed into a `BinaryBitmap` format. Then, the `decode` method from the `MultiFormatReader` class is invoked to decode and subsequently retrieve the `UID` associated with the ticket.

To display the QR code within the mobile application, users can select the relevant ticket from their list of purchased tickets, as specified in `ListBoughtTickets (UC 20)` in Section 3.2.2.


```

string ScanQrCode(string path)
{
    System.Drawing.Image image;
    int width, height;
    using (var fileStream = new FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read))
    {
        using var imageStream = System.Drawing.Image.FromStream(fileStream, false, false);
        image = imageStream;
        height = image.Height;
        width = image.Width;
    }

    var bin = File.ReadAllBytes(path);
    //make a bmp from the selected file
    using (var stream = new MemoryStream(bin))
    using (var bmp = new System.Drawing.Core.Bitmap(stream))
    {
        var source = new BitmapLuminanceSource(bmp);
        var bitmap = new BinaryBitmap(new HybridBinarizer(source));
        var result = new MultiFormatReader().decode(bitmap);
        return result?.Text;
    }
}

```

Figure 5.15: The `ScanQrCode` method implemented with the `Zxing` library

To streamline the user experience by avoiding additional screen creation, in this and some other scenarios pop-up overlays were adopted. The `Mopups` *NuGet* package offers high-level components designed for crafting highly customizable pop-ups and is excellent for addressing this requirement.

Specifically, the `mopups:PopupPage` class was employed, in the place of the standard `ContentPage` class. Also, the `MopupService`, was registered as a singleton realization of the `IPopupNavigation` interface within the dependency injection container, facilitating stack-like navigation for these pop-up elements. A demonstration of this implementation is depicted in Figure 5.16.



Figure 5.16: The popup delivered with the `Mopups` library

To conclude this section, attention is drawn to the implementation of statistical charts in the `StatisticsView` and `StatisticsViewWin` classes. The `LiveChartsCore` *NuGet* package offers many customizable and animated charts built upon Google's *Skia* open-source graphics library. `CartesianChart`, `PieChart` and `PolarChart` were used to display the `ISeries` instances populated inside the view model with the pertinent information.

A notable feature of these charts is the ability for dynamic value updates based on real-time data changes. As discussed above, this functionality binds well with the *Firestore Realtime Database* service, which can provide real-time notifications of data changes. Such changes could be triggered by various customer events, like ticket purchases or completion of questionnaires.

5.3 Database seeders

During the first debugging sessions for the application, it was realised that populating the data model was a critical priority. This need is addressed by the implementation of the singleton class `DbPopulatorManager`, designed to populate various model classes. For the `Artwork` and `Exhibit` instances, the museum's website serves as a reliable source for gathering accurate information. In contrast, `Customer`, `Questionnaire`, and `Ticket` instances are generated randomly with a few statistical policies and then appropriately interlinked. The manager's methods are then invoked to populate the remote database with this synthetic data.

This approach facilitated a consistent simulation experience that aligned with user expectations throughout the various stages of screen development. Moreover, it guaranteed a consistent availability of test data for the validation of emerging functionalities.

5.4 Issues encountered

As one might expect with an emerging technology like the .NET MAUI framework, certain challenges were encountered when diving into its more specialized features. This is an expected hiccup for any new framework, but, fortunately, its active community support often provides temporary solutions, as many issues have already been identified by other developers.

Therefore, throughout the development phase, it was occasionally imperative to consult the "Issues" section of the MAUI repository, accessible at <https://github.com/dotnet/maui/issues>, in order to find workarounds for the problems encountered.

The focus of this discussion is not to enumerate exhaustively every resource consulted, but rather to spotlight the most salient challenges encountered and the strategies implemented to address them.

The overall impression suggests that while the framework aims to support compilation for both desktop (Windows and Mac) and mobile (Android and iOS) platforms, its focus appears to be more skewed towards mobile development. This assessment is primarily based on the problems encountered in achieving full cross-platform functionality, particularly for the staff application.

Previously, in the Xamarin era, these platform-agnostic features were bundled under the `Essentials` namespace. In MAUI, they are distributed across different domains such as `Devices.Sensors`, for sensor-related information, and `Media`, for multimedia data capture [Leibowitz, 2022]. With regard to the latter, challenges were encountered with the `MediaPicker` class, specifically when attempting to scan QR codes through the webcam. An effective workaround for this issue has been proposed in Issue #7660 by Giampaolo Gabba, involving a partial, custom implementation of the `IMediaPicker` interface. This custom class is then registered in the dependency injection container as a singleton service via the command `AddSingleton<IMediaPicker, CustomMediaPicker>()`.

However, while the workaround made it feasible to capture single snapshots with the webcam, there was no solution for capturing a continuous webcam feed for real-time QR code

scanning. Neither does MAUI offer built-in support for this feature, nor there is a suitable *NuGet* package available to accomplish this specific functionality. The only approach found is to have the user manually capture a photo using the `CustomMediaPicker` service, and then, once the photo is saved in a temporary file, its contents can be decoded using the *ZXing* library, as previously depicted in Figure 5.15.

```
var width = 1360;
var height = 920;
Microsoft.Maui.Handlers.WindowHandler.Mapper.AppendToMapping(nameof(IWindow), (handler, view) =>
{
    var mauiWindow = handler.VirtualView;
    var nativeWindow = handler.PlatformView;
    nativeWindow.Activate();
    IntPtr windowHandle = WinRT.Interop.WindowNative.GetWindowHandle(nativeWindow);
    WindowId windowId = Microsoft.UI.Win32Interop.GetWindowIdFromWindow(windowHandle);
    AppWindow appWindow = Microsoft.UI.Windowing.AppWindow.GetFromWindowId(windowId);
    var display = DeviceDisplay.Current.MainDisplayInfo;
    appWindow.MoveAndResize(new RectInt32((int)(display.Width / 2 - width / 2),
        (int)(display.Height / 2 - height / 2), width, height));
    appWindow.TitleBar.ExtendsContentIntoTitleBar = true;
});
```

Figure 5.17: The custom implementation of the `IWindow` handler

Handling window resizing and repositioning was another challenge. This is particularly useful to ensure all information is displayed in an aesthetically pleasing and functional manner. The framework does not offer out-of-the-box support for setting a minimum window size. As a result, custom handlers have to be resorted to to fulfil this requirement. Handlers can be customized to augment the appearance and behaviour of a cross-platform control beyond the customization that is possible through the control’s API. This customization, which modifies the native views for the cross-platform control, is achieved by modifying the mapper for a handler [Britch, 2022a]. As depicted in Figure 5.17, the method `MoveAndResize()` from the `AppWindow` class was utilized. This was achieved after obtaining the handle associated with the window. This approach enabled the customizing of window dimensions to meet the requirements for optimal information display.

There have been a few challenges with the dynamic updating of collection views in the staff application, especially notable in screens where the displayed data are subject to frequent changes, such as on the chat page. Occasionally, page layouts fail to dynamically resize in response to window adjustments, and the collection views struggle to refresh after the addition of a new message. These issues have been discussed in several GitHub issues, including #6959, #8636 and #8534. To date, no definitive solution has been found. However, in this context, a temporary workaround has been to provide users with the option to manually refresh the page through a designated button, to be used in the rare instances when this problem manifests.

Lastly, challenges were also faced with the *Picker* component, particularly when dynamically updating its items source. A viable workaround for this problem can be found in GitHub issue #9739, which suggests setting the *Picker*’s content to *null* before re-assigning it, to trigger garbage collection. Interestingly, this approach has also proven useful in other contexts where forcing a visual component to refresh is necessary. At times, instead, special attention had to be given to the hierarchical arrangement of visual components. This was crucial to prevent parent containers, like grids, from hindering their child elements from refreshing, as was the case with *ScrollView* components.

This paragraph details the proper usage of the software for both staff members and customers. The manual offers concise instructions to complete tasks such as purchasing or validating tickets and filling out questionnaires. Adherence to the guide ensures the software's optimal utilization and aids in resolving potential issues.

6.1 Staff application usage

The staff application serves as a management tool for museum employees. It is primarily utilized for validating customer tickets and overseeing the internal storage. To ensure security, only museum computers have access to the app. Furthermore, an existing account is required to log in too.

6.1.1 Installation and setup

The software is presently accessible for Windows 10, version 1709 and onwards [Microsoft, 2023], and can be downloaded from <https://github.com/MrPio/MuseoOmeroApp-MAUI/releases/tag/Windows>.

As the application has not been published on the *Microsoft Store* or any other official store, it must be installed as a sideload application.

The first step is to enable developer mode in the Windows *Settings* application. This can be achieved by navigating to *Update & Security > For developers* and ensuring that the *Developer mode* or *Sideload apps* setting is checked, as illustrated in Figure 6.1.

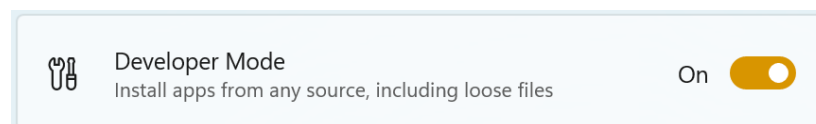


Figure 6.1: Developer mode is required for the installation of sideload applications

Then, the custom certificate is required. As shown in Figure 6.2, to install the certificate, one should double-click on the `.cer` file from the previously downloaded archive and select the option to *Install Certificate*. Then, the *Local Machine* option should be selected, and the certificate should be imported into the *Trusted Root Certification Authorities* folder.

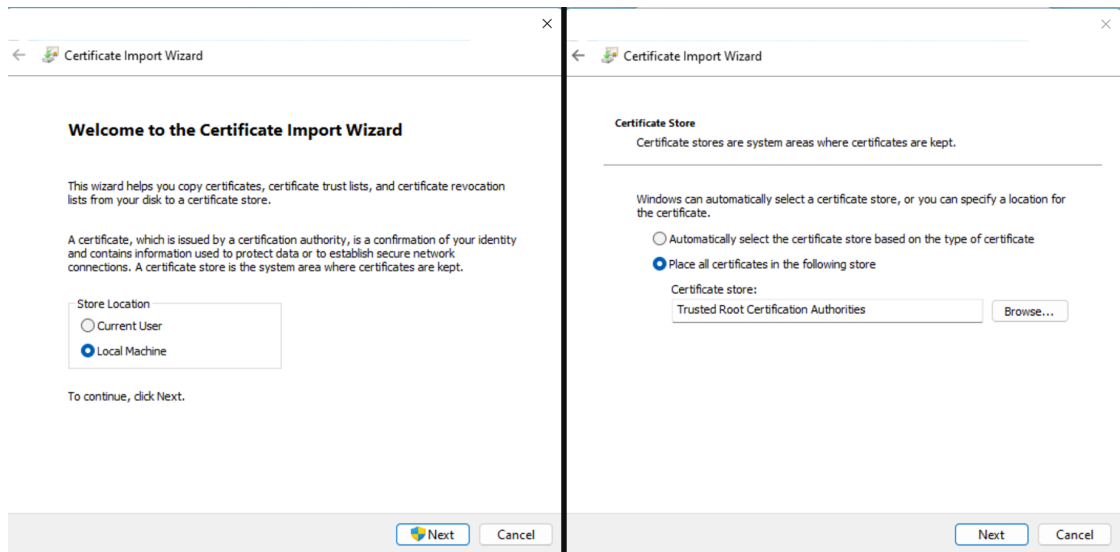


Figure 6.2: The procedure to follow to install the custom certificate

The `.msix` file can be installed using the *Msix Packaging Tool* and following the wizard, as shown in the figure 6.3. Alternatively, the `.msix` file can be created using the Windows PowerShell command `Add-AppxPackage` if the GUI tool is inaccessible. For security purposes, it is recommended to disable the developer option after installing the application to prevent the installation of potentially unwanted applications on the machine.

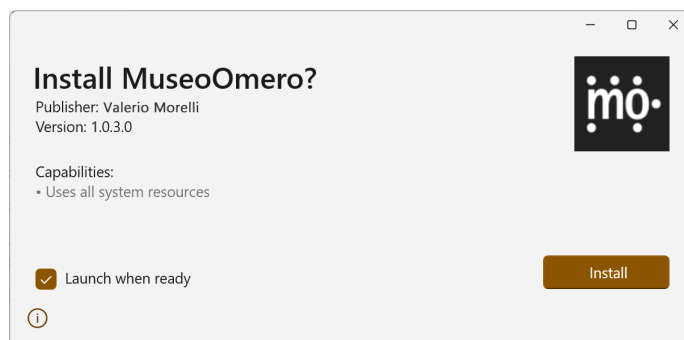


Figure 6.3: The installation of the *msix* package for the staff application

6.1.2 Authentication

After installation, the login page will appear during the software's initial execution. The login page can be seen in Figure 6.4. At this point, the employee is required to enter their login information.

It's important to note that an account creation option is not available, as this responsibility is entrusted to the administrator who creates accounts on an as-needed basis through the Firebase *Authentication* dashboard. However, for testing purposes, you may use a test account with the email `museoomero@test.com` and the password `museoomero`.

It is also possible to recover an account's forgotten password from this page by clicking on the designated button. The employee will then receive an email containing instructions on how to proceed to the provided address. After completing the operation, the newly created password can be used to log in with the email address associated with the account.

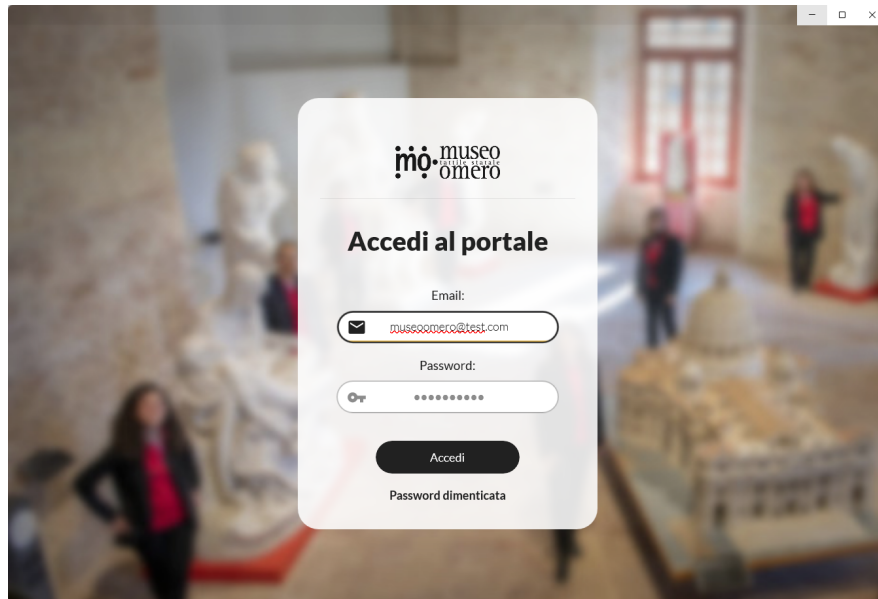


Figure 6.4: The staff application login page

After successfully logging in, the software will store the session, allowing the account to be accessed in future without requiring any user action. To switch the account, one can exit from the current one by using the designated button on the top bar menu, as indicated in Figure 6.5.



Figure 6.5: The log-out button on the top menu of the staff application

6.1.3 Core Functionalities

As can be seen in Figures 4.4 - 4.9, the application is divided into six different screens navigable through the lateral slide menu. In details, they are:

- *Home*: This is the main page displayed after logging in successfully. It exhibits a dashboard of the most recent information with a synopsis of various internal management aspects, such as today's ticket and questionnaire details, the number of unread messages, and the overall visit type average. Furthermore, a rapid overview of all owned artworks separated per room is located in the lower-left corner of the display. Clicking on these summary boxes will redirect you to their corresponding dedicated pages.
- *Statistics*: This page displays a collection of statistics primarily dedicated to questionnaire compilation, but also related to ticket purchases, visits and online artwork views. There are various types of charts, like the one shown in Figure 6.6. It is worth noting that the statistics period range is set to the most recent three months as default, but it can be edited using the start and end date pickers.
- *Ticket office*: On this screen, the employee has the ability to validate purchased tickets, view their relevant information and purchase new tickets for registered customers.

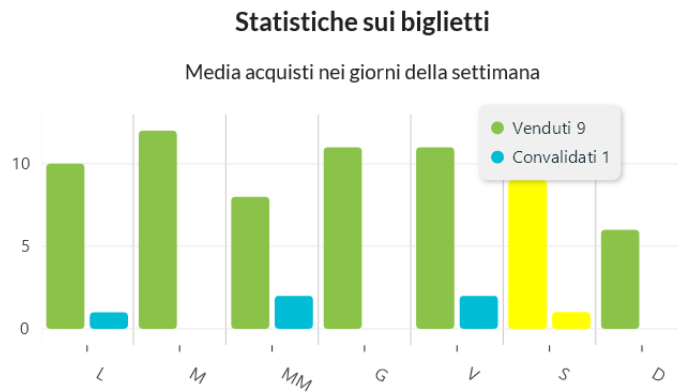


Figure 6.6: The statistics on the weekly ticket validation purchase ratio

Additionally, the page displays a summary of all sold tickets on the right-hand side, which can be filtered by validity date, ticket type, purchase date or validation date. In case of selection, a popup with a summary of the ticket information, including its QR code, will appear. The employee may validate the ticket by simply clicking the button, given that it is valid. However, scanner validation is also possible using the webcam to scan the QR code. For this purpose, the button on the left panel can be used to start capturing the webcam feed, as shown in Figure 6.7.

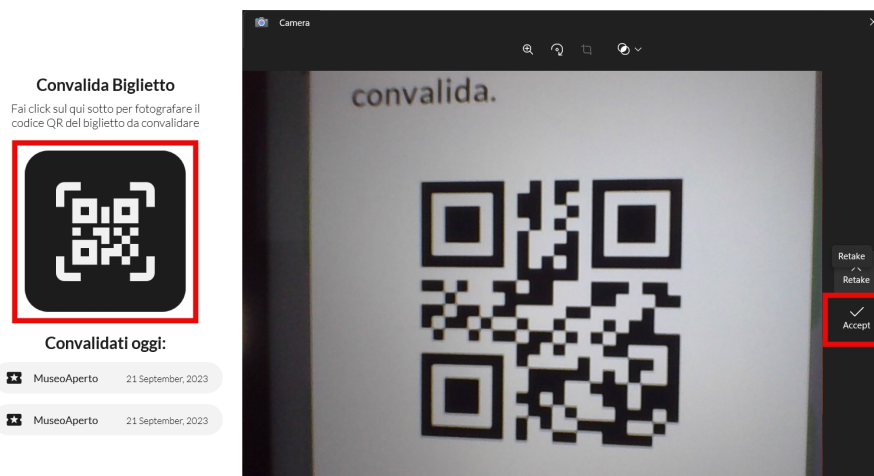
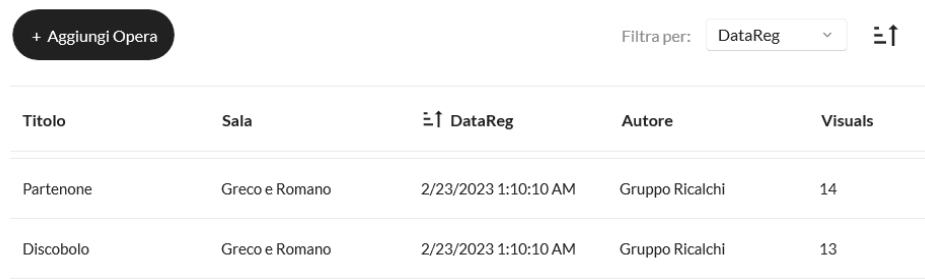


Figure 6.7: The procedure for scanning a QR code

- *Artworks and exhibits:* This screen is for storing artwork and exhibition information. The staff user may add new entries, alter existing ones, and view all associated details. The two lists can be sorted by selecting a field in the picker at the top of the screen or by clicking on the column header, as illustrated in Figure 6.8. To switch between managing artwork and exhibits, the user can employ the two distinct navigation buttons located at the bottom of the page. Additionally, employees may store new entries by clicking the buttons located on the top left and filling out the provided form. If the attempt to save is unsuccessful, a message indicating an input error during compilation will be displayed. Finally, selecting an item from the list will prompt the presentation of a form containing a summary of the artwork or exhibit details. If the selected item is an exhibition, it is possible to revise certain aspects, such as its presentation picture and the list of featured artworks.



Titolo	Sala	DataReg	Autore	Visuals
Partenone	Greco e Romano	2/23/2023 1:10:10 AM	Gruppo Ricalchi	14
Discobolo	Greco e Romano	2/23/2023 1:10:10 AM	Gruppo Ricalchi	13

Figure 6.8: The artwork collection

- *Chats:* The chat screen serves as a means to communicate with customers. The page displays two sections, depicted in Figure 6.9. The left part of the screen allows the

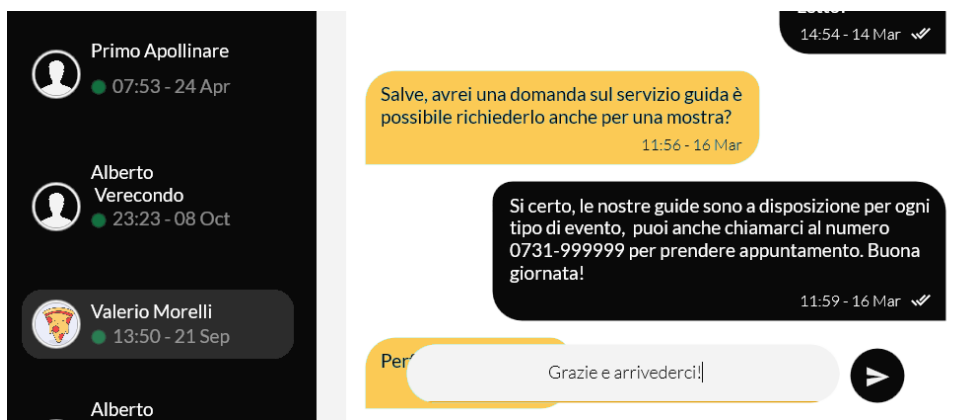


Figure 6.9: The chat page

staff user to select the customer whose chat appears on the right part. Any user who has sent the museum at least one message should appear on the left side with their respective avatar, full name, and most recent access time. The staff member is able to send messages through the chat of the chosen user and observe the read status of their messages. This is indicated by an icon at the bottom of each message, either single or double thick, signifying whether the user has read the message. The chat updates automatically, ensuring that any response from the customer is promptly displayed if their chat is open.

- *Account:* This screen presents an overview of all the account details. The staff member can modify data such as the master data, account password, and profile picture. The password-changing process is identical to that detailed on the login page. After making the desired changes, the user must click the save button on the top to apply them.
- *Settings:* This page contains options for customising the application’s behaviour. At present, only one setting is available, namely the theme switching, depicted in Figure 6.10. Once selected, the application’s colours will change in accordance with the palette specifications defined in Figure 5.10.

6.2 Customer application usage

The museum’s customer application serves as a communication tool between customers and the services provided by the institution. To use the application, a valid account is required,

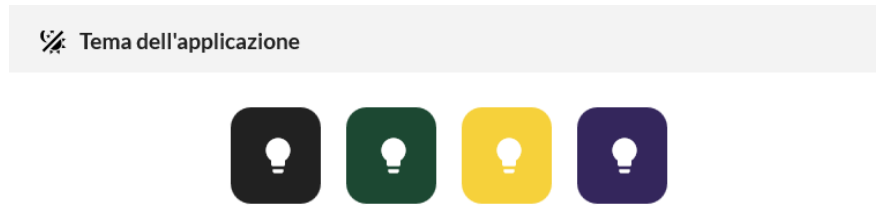


Figure 6.10: The theme selection buttons on the settings screen

which can be created when the application is launched.

6.2.1 Installation and setup

The requirements for running the application include having Android 5.0 or a higher version.

At present, the application is not available on the *Play Store*, but it can be downloaded, as a `.apk` file, from <https://github.com/MrPio/MuseoOmeroApp-MAUI/releases/tag/Android>.

For security reasons, the Android operating system prevents users from installing applications downloaded from unofficial sources. That being said, it is necessary to grant permission by going to *Settings > Security* and checking the *Unknown sources* option.

However, starting with Android 8.0, app installation permissions were tightened. Instead of allowing or disallowing unknown sources globally, the permission should be managed on a per-app basis as shown in Figure 6.11.

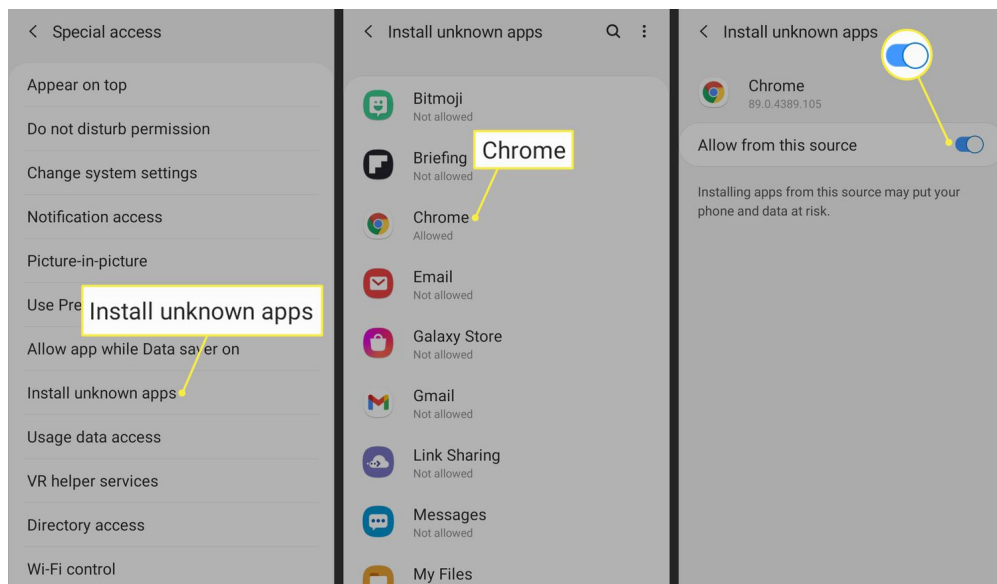


Figure 6.11: Enabling app installations from unofficial sources on Android 8.0 and onwards

Once permission has been granted, the application can be installed by running the `.apk` file through the system installer.

6.2.2 Authentication

The initial screen displayed is the login page. The user has the option to enter their existing account details or establish a new account by selecting the respective button displayed at the

bottom of the login page. Both screens are illustrated in the screenshots shown in Figure 6.12. To create an account the email, username, password, and password confirmation fields, which are marked with an asterisk symbol on the right-hand side, must be filled in. Furthermore, the user may provide their first and last names, as well as mobile phone numbers, as optional details. If the user has an account associated with their email address but has forgotten the

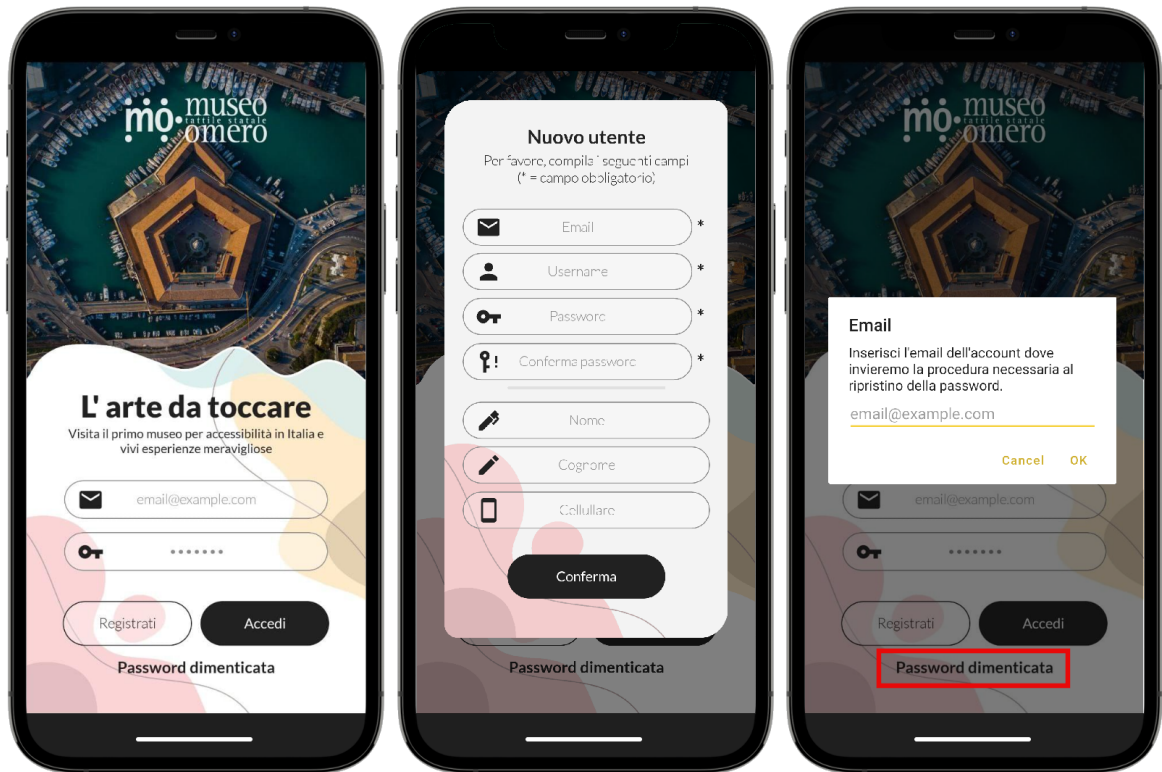


Figure 6.12: The customer application authentication screens

password, they can change it by clicking on the designated button and following the guided procedure.

Upon successful login, the system will store the session and automatically remember the active account to shorten future application access time. Nevertheless, the user can log out at any time by using the logout button provided in the top bar menu, highlighted in Figure 6.13.

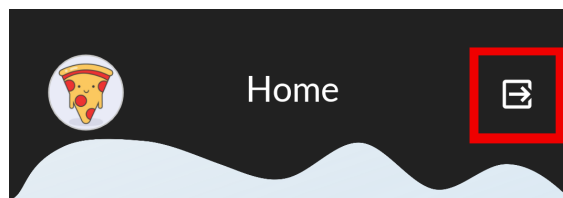


Figure 6.13: The logout button on the top bar of the customer application

6.2.3 Core Functionalities

As depicted in Figures 4.10 - 4.12, the application is subdivided into five main screens navigable through the bottom navigation bar plus three pages dedicated to the account, artworks and exhibitions overview and questionnaires compilation. These screens are detailed in the following:

- **Home:** The initial screen upon accessing the application is the home page, which is split into two vertical sections. These sections display carousel views of the artworks and exhibitions respectively. The artworks section is subdivided based on the room in which they are currently displayed, and this filter can be chosen using the top radio buttons as depicted in Figure 6.14. On the other hand, the exhibitions section presents both past and present exhibits in chronological order.

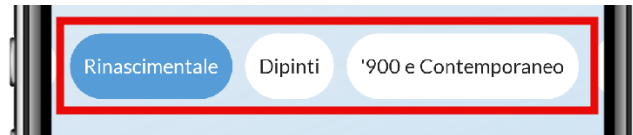


Figure 6.14: The room selection buttons

When an artwork or exhibition element is selected, the application will transition to a page displaying its complete information. These two screens are illustrated in Figure 6.15. If the selected item is an artwork, the visual counter of that artwork will be incremented by the system and displayed at the bottom of its overview page.



Figure 6.15: The artworks and exhibits overview pages

- **My tickets:** This page presents a chronological list of all tickets purchased by the user. For instance, a newly purchased ticket from the *Ticket office* screen will appear in this list and be readily accessible. As shown in Figure 6.16, each ticket element's appearance varies depending on its validity status. Users can filter the list by the ticket's validity month, by selecting a date on the filter picker located at the top of the page, depicted in Figure 6.16.

Upon selecting a ticket item, a popup window will display a summary of the ticket information, including a QR code required for validation.



Figure 6.16: The bought tickets list's filter

- *Ticket office*: This page is dedicated to ticket purchasing and presents three distinct buttons, one assigned to each ticket category: open museum, exhibition, and workshop tickets. Once selected, each button expands to reveal three purchase options, namely solo visits, group visits, and guided visits. The ticket price is exclusively determined by the ticket category and is represented by the euro symbol, as illustrated in Figure 6.17.

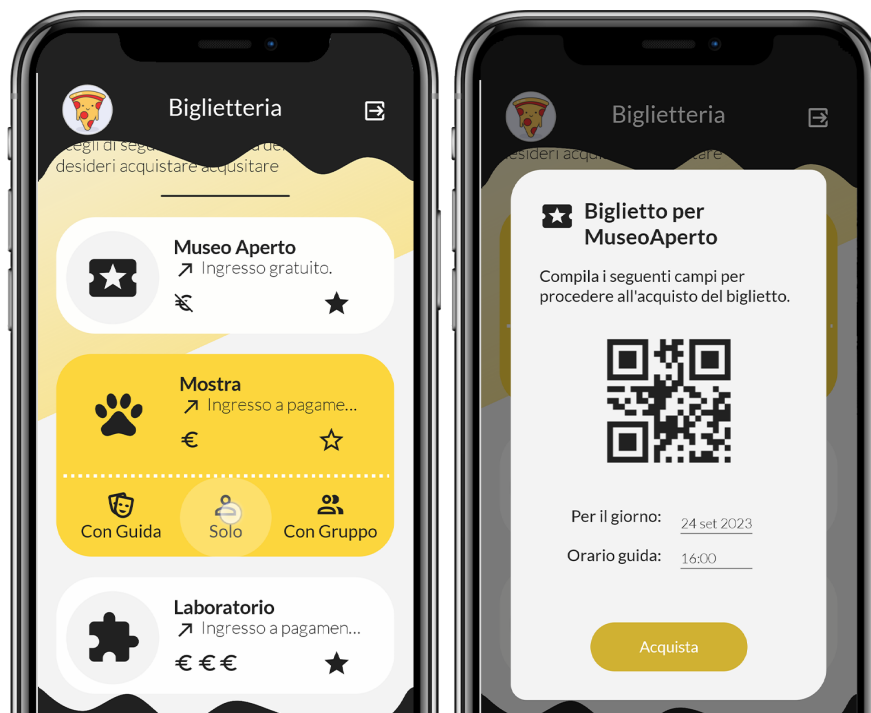


Figure 6.17: The ticket office purchase procedure

When purchasing a ticket, the user must specify the date of their visit as the ticket is only valid for that specific day. If the user wishes to purchase a ticket for an exhibit, they must also select the desired exhibit. Once the required information is provided, a confirmation message with the purchase details is displayed, including the final price.

If the user confirms the details, the ticket will be added to their bought ticket list.

- *Chat*: This page enables communication between the customer and the museum. Upon sending the first message, a new chat is created for the specific customer’s account in use. The chat may be used to seek assistance with any inquiries about the available services and their fruition. As the chat accumulates messages over time, the user might need to filter for a keyword to locate an older query or response. To achieve this, the filter entry at the top of the page can be utilised, as shown in Figure 6.18. Once activated, the filter will exclusively display messages that match the provided string pattern, which will be emphasised within the displayed messages.



Figure 6.18: The customer’s chat filter feature

- *Questionnaires and statistics*: This screen provides an outline of the museum’s visitation history. It comprises of two sections, featured in Figure 6.19.

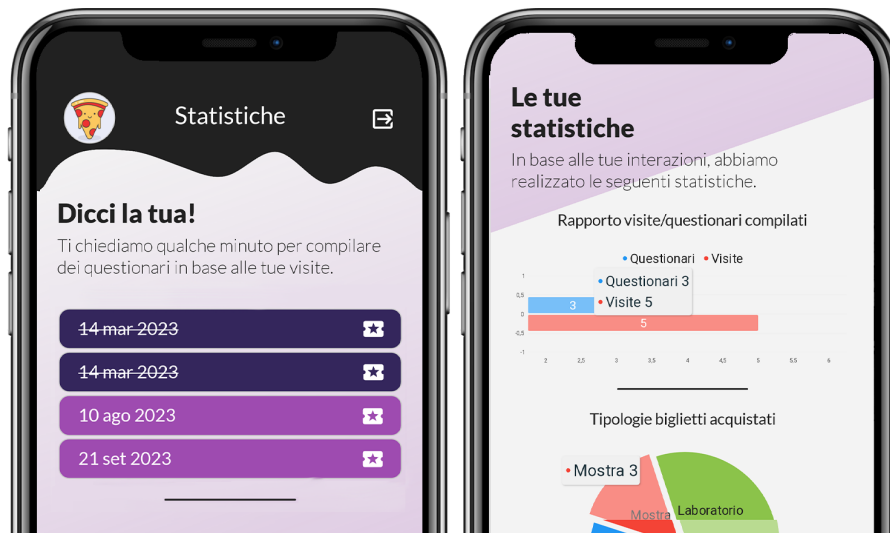


Figure 6.19: The customer’s personal statistics page

The first section illustrates past visits in a list format, which appearance is dependent on the questionnaires’ completion stage. If the questionnaire has not already been filled in for recent visits, it is possible to do so by clicking on it. The application will then navigate to a dedicated compilation page, as depicted in Figure 6.20. Once the user has

chosen an answer from those available for each field, and confirms the operation, the compilation will be stored and the corresponding visit will be updated.

Figure 6.20: The questionnaire’s compilation page

In the second section of the page the client can review their personal statistics. At the top, there is a bar chart displaying the ratio of completed questionnaires to visits, while the bottom exhibits a pie chart depicting the different types of purchased tickets.

- *Account:* This page presents a summary of the current account’s information. To access this page, one should click on the avatar image located on the top left-hand side of the previously discussed screens. Figure 6.21 depicts how the page shows a list of account information that the user can modify. Additionally, it hosts a few useful

Figure 6.21: The customer’s account page

information relating to interaction with the museum’s services. The user may also change their profile picture by clicking on it and selecting a new image from their gallery. In conclusion, the user can save the changes by clicking on the designated button at the bottom.

In this chapter, we will analyse the work carried out and draw appropriate conclusions. We will also emphasise lessons learned, specifically those poor decisions made in the development stages due to lack of experience that should be noted to prevent their recurrence in the future.

7.1 Obtained results and implications

The current thesis covers the development and implementation of museum management software adopting the emerging MAUI technology. The project requirements indicated the necessity of two separate applications: one for internal management, to be used by museum staff, and the other to be provided to customers. After a thorough requirements analysis and design process, a robust framework was established. The coding of the model and interface classes for the Firebase services, as well as the distinct component screens for both applications, were then developed around it. In conclusion, a comprehensive guide has been supplied for documenting the installation and usage of the software.

The implementation of this system carries multiple implications. First of all, it represents a museum's modernisation attempt and an improvement in its services. The lack of appropriate internal management tools resulted in the staff's dependence on insecure methods, such as exchanging voice messages or inadequately organised spreadsheets. These practices have compromised the quality of the services offered and are likely to result in long-term errors and delays.

Conversely, the absence of a mobile app may indicate that the museum is unable to keep pace with modern trends. This could potentially make customers feel uncomfortable when using the services provided due to the lack of modern conveniences.

The newly developed system facilitates customers to purchase tickets from the comfort of their own sofa and to provide feedback regarding their visit via a digital questionnaire while on the way home, as opposed to making a reservation by call and completing tedious paper forms manually within the facility.

As a centralised system, staff can access up-to-date information at all times by consulting the application on their monitor. This enables them to anticipate bookings in the day and organise guides' shifts to manage them. With the appropriate permissions, it will be possible to handle internal information digitally, thus avoiding the need for communication between different departments. By making changes directly to a centralised database, data inconsistencies can thus be avoided.

7.2 Future improvements

The produced software serves as a solid foundation for creating a dependable management system, but it is not a final solution. The introduction of .NET 8 LTS and the growth of MAUI brought numerous advantages that the software should leverage.

With regards to additional features, it might be useful to use the real-time functionality of Firebase *Realtime Database* further. Currently, this feature is only being implemented for chat management purposes. One potential improvement would be to notify the staff application when a ticket is purchased remotely or a questionnaire is completed. Similarly, customers could receive a push notification when a new exhibition is scheduled.

Additionally, the present software does not enable effective laboratory activity management for both administration purposes and ticket selection. Furthermore, it does not provide customers with the option to sign in and log in using their *Gmail* accounts.

It may be worth considering the possibility of introducing new features not currently available in the museum as part of future developments. For example, a rewards system could be implemented, enabling customers to earn points through buying tickets or completing questionnaires.

Overall, the software offers numerous functions that meet the demands of the museum, although each may require more detailed attention to detail and management.

7.3 Learned lessons

This project was significant in gaining fundamental knowledge of application development in the .NET domain. Understanding MAUI development equates to comprehending the entirety of .NET, as widely discussed in Chapter 1, since different Microsoft technologies share a common codebase. In short, if the need for Xamarin, Blazor or ASP NET projects arises, the knowledge obtained here will prove invaluable. For instance, familiarity with the C# language and its libraries, as well as proficiency in using the IDE, will be necessary.

At the conclusion of a project, it is advisable to evaluate the process for errors, to gain insight and thus avert their repetition during forthcoming ventures.

Looking back at the initial stages of development, various trials and experiments through sketches were realized to comprehend how the framework operates. As some of these were related to specific application screens or components, it is evident that the initial classes contained lower-quality code compared to the more recent ones, created with a better understanding of the platform. On several occasions, non-trivial problems resulted in over-complicated and sometimes ineffective solutions, which could have been substituted with more graceful ones based on aspects not yet studied. Essentially, it would be advisable to enhance one's familiarity with a new platform before developing elements of the final software, regardless of its potential robust design.

Another frequent issue concerns insufficiently detailed requirements leading to wasted efforts in developing features that do not correspond to actual needs. For example, time was spent creating a validation feature for the customer's application, only to discover that it was unnecessary for the customer to independently verify their purchased ticket. This led to a pointless waste of time that could have been prevented with better planning.

Moreover, excessive time ought not to be expended in searching for a solution to an apparently complex issue concerning a rather specific feature, driven by the desire to complete that task without leaving it unfinished. The knowledge gained from experience dictates that it is prudent to defer the resolution of the said problem to a later moment, to avoid any distraction from the primary objective. This is because there is a high chance that, with a deeper understanding, one can identify a more straightforward solution.

Bibliography

- ADAM (2019), «Functional Model-View-Update Architecture for Flutter», <https://buildflutter.com/functional-model-view-update-architecture-for-flutter/>. (Cited at page 14)
- ANDY DE GEORGE, P. P. (2022), «XAML Syntax In Detail», <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/xaml-syntax-in-detail?view=netframeworkdesktop-4.8>. (Cited at page 9)
- BANDT, T. (2020), «Model-View-Update (MVU) – How Does It Work?», <https://thomasbandt.com/model-view-update>. (Cited at page 15)
- BILLWAGNER, G. I. G. Y. M., FLASHOVER (2022), «XAML», <https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview>. (Cited at page 9)
- BRITCH, D. (2022a), «Customize controls with handlers», <https://learn.microsoft.com/en-us/dotnet/maui/user-interface/handlers/customize>. (Cited at page 68)
- BRITCH, D. (2022b), «Publish and subscribe to messages», <https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/messagingcenter?view=net-maui-7.0>. (Cited at page 16)
- BRITCH, D. (2022c), «XAML», <https://learn.microsoft.com/en-us/dotnet/maui/xaml/?view=net-maui-7.0>. (Cited at pages 8 e 9)
- BRITCH, D. (2022d), «XAML markup extensions», <https://learn.microsoft.com/en-us/dotnet/maui/xaml/fundamentals/markup-extensions?view=net-maui-7.0#main>. (Cited at page 9)
- BRITCH, D. (2023a), «Add images to a .NET MAUI app project», <https://learn.microsoft.com/en-us/dotnet/maui/user-interface/images/images>. (Cited at page 61)
- BRITCH, D. (2023b), «Binding value converters», <https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/data-binding/converters?view=net-maui-7.0>. (Cited at page 20)
- BRITCH, D. (2023c), «Data binding», <https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/data-binding/?view=net-maui-7.0>.

- BRITCH, D. (2023d), «Data binding basics», <https://learn.microsoft.com/en-us/dotnet/maui/xaml/fundamentals/data-binding-basics?view=net-maui-7.0>. (Cited at page 16)
- CAMUSO (2022-2023), «Sviluppare app Android/iOS con .NET MAUI», https://www.youtube.com/playlist?list=PL0qAPtx8YtJfLud-Kz7_iZWSqitVLYjxj, [Online; accessed 23-April-2023].
- GABRANG, C. (2022), «Custom Fonts Material Design Icons in .NET MAUI», .
- GEORGE, A. D. (2022), «Attached Properties Overview», <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/attached-properties-overview?view=netframeworkdesktop-4.8>. (Cited at page 9)
- GOLDMAN, M. (2022), *.NET MAUI in Action*.
- GOLDMAN, M. (2023), *.NET MAUI Cross-Platform Application Development*. (Cited at page 7)
- GRASSINI, A. (2015), *Per un'estetica della tattilità*.
- JEAN-MARIE (2022), «MVU or MVVM with MAUI App?», <https://www.jmparent.com/2022/01/18/mvu-or-mvvm-with-maui-app/>. (Cited at page 15)
- LEIBOWITZ, M. (2022), «Modernize Essentials Namespaces», . (Cited at page 67)
- MENDES, A. (2023), «Single page applications - the future of web applications», . (Cited at page 63)
- MICROSOFT (2023), «MSIX features and supported platforms», <https://learn.microsoft.com/en-us/windows/msix/supported-platforms>. (Cited at page 69)
- MINISTRYOF CULTURE (2016), «Museo tattile statale Omero», . (Cited at page 22)
- MONTEMAGNO, J. (2022), «.NET MAUI for Beginners», <https://www.youtube.com/watch?v=KmlQLSKqvvi&list=PLwOF5UVsZWUjNR3roRK79QgBcKLYOX48I>, [Online; accessed 23-April-2023].
- NETWORKMUSEUM (2017), «Con gli occhi e con le mani», . (Cited at page 21)
- PEDRI, S. (2022), «Announcing .NET Community Toolkit v8.0.0 Preview 3», <https://devblogs.microsoft.com/ifdef-windows/announcing-net-community-toolkit-v8-0-0-preview-3/#partial-property-changed-methods>. (Cited at page 14)
- SHAUN LAWRENCE, G. V. (2022), «C Markup», <https://learn.microsoft.com/en-us/dotnet/communitytoolkit/maui/markup/markup>. (Cited at page 10)
- SHVETS, A. (2019), *Dive Into Design Patterns*. (Cited at page 15)
- SUÁREZ, J. (2022), «UI Challenges», <https://www.youtube.com/watch?v=Oqst00uLTgU&list=PLQPLL9Pbjp33gIEmc-JH5rsBiaalJj0wx>, [Online; accessed 23-April-2023].
- VERMEIR, N. (2022), *Introducing .NET 6*. (Cited at page 12)
- VERSLUIS, G. (2022a), «Essential plugins for your .NET MAUI», <https://www.youtube.com/watch?v=nCNh9G-Q688&list=PLfbOp004UaYVgzmTBNVI0q12qF0LhSEU1>, [Online; accessed 23-April-2023].

- VERSLUIS, G. (2022b), «Learn to Build Your First Mobile App: .NET Maui Crash Course», <https://www.youtube.com/watch?v=mgW6xviirQk&list=PLfbOp004UaYVt1En4WW3pVuM-vm66OqZe>, [Online; accessed 23-April-2023].
- VERSLUIS, G. (2022-2023), «.NET MAUI 101 - The Fundamentals of .NET MAUI», https://www.youtube.com/watch?v=U0K1IlQfzJY&list=PLfbOp004UaYWMhAu5zy7bkUrAPGD_I_2-, [Online; accessed 23-April-2023].

Websites consulted

1. Microsoft Learn - MAUI documentation – <https://learn.microsoft.com/it-it/dotnet/maui/?view=net-maui-7.0>
2. Wikipedia - SOLID – <https://en.wikipedia.org/wiki/SOLID>
3. Wikipedia - Alonzo Church – https://en.wikipedia.org/wiki/Alonzo_Church
4. Wikipedia - Functional programming – https://en.wikipedia.org/wiki/Functional_programming
5. Wikipedia - Side effect (computer science) – [https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))
6. MicrosoftFormums - InitializeComponent before set BindingContext or viceversa? – <https://social.msdn.microsoft.com/Forums/en-US/9b523e05-a6b8-442d-b0ec-5f3def6c5be0/initializecomponent-before-set-bindingcontext-or-viceversa?forum=xamarinforms>
7. Reddit - Binding doesn't work when passed to ConverterParameter – https://www.reddit.com/r/xamarindevelopers/comments/tld0lq/binding_doesnt_work_when_passed_to/
8. Wikipedia - Museo Omero – https://it.wikipedia.org/wiki/Museo_Omero

Acknowledgements

I would like to express my gratitude towards my parents for their unwavering trust and support, both financially and morally, throughout my academic journey.

Additionally, I extend my sincere appreciation to all who have contributed to completing this project and writing this thesis. In particular, I would like to thank my supervisor, Professor Domenico Ursino, for his invaluable guidance in correcting the thesis and his endless availability. I would like to express my appreciation also to my co-supervisor, Doctor Enrico Corradini, for his valuable insights in the course of this project, always readily available.