POLISH-JAPANESE ACADEMY
OF INFORMATION TECHNOLOGY

Faculty of Information Technology

Chair of Software Engineering

Software and Database Engineering

Piotr Kwiatkowski
Student no. 16085

Security for modern mobile applications

Master of Science Thesis written
under supervision of:

Mariusz Trzaska Ph. D.

Warsaw, June 2019

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Piotr Kwiatkowski
Nr albumu 16085

Zabezpieczenia w nowoczesnych aplikacjach mobilnych

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, Czerwiec 2019

Abstract

Mobile phones and tablets are one of the most popular devices used by people all over the world. Due to this fact the number of applications targeting devices is still growing. Such applications address the wide spectrum of interests, starting from productivity area, through banking ending with the games industry.

However, no matter what kind of mobile application is being built, it almost always requires some level of security. Development of such application in a secure manner is usually difficult as the number of frameworks and protocols proposed for this purpose is significant. On top of that, the majority of developers are not familiarized with developing in a secure way, not to mention thinking about it at all.

This thesis discusses how to easily develop modern and secure mobile applications and facilitate this process to those who are not accustomed to security areas. It also briefly presents a state of the art and both advantages and disadvantages of existing solutions used worldwide.

The outcome of this work is a concept of a cross-platform framework that enables simple set-up of security issues for every mobile application. Moreover, it guarantees easy modifications or extensions of default functionalities provided by the framework, depending on the needs. As part of the thesis a prototype framework has been created. In order to prove that it works correctly, an exemplary mobile application has been built as well.

Keywords: Information Technology, mobile applications, security, cross-platform.

Streszczenie

Telefony oraz tablety są jednymi z najbardziej popularnych urządzeń używanych na całym świecie. Z tego względu liczba aplikacji mobilnych wciąż rośnie. Rozwiązania tego typu obejmują bardzo szeroki zakres zainteresowań, zaczynając od obszarów produktywności, poprzez bankowość, a kończąc na przemyśle gier.

Jednakże, niezależnie od tego jaki rodzaj aplikacji mobilnej jest budowany, prawie zawsze wymaga ona pewnego poziomu zabezpieczeń. Tworzenie takich aplikacji w bezpieczny sposób jest zazwyczaj skomplikowane ze względu na to, że liczba proponowanych dla tego celu bibliotek oraz protokołów jest znacząca. Dodatkowo, większość programistów nie jest zaznajomiona z programowaniem w bezpieczny sposób, nie wspominając o myśleniu o tym jako całości.

Ta praca omawia, jak łatwo tworzyć nowoczesne i bezpieczne aplikacje mobilne oraz ułatwić proces ich implementacji dla tych, którzy nie są przyzwyczajeni do obszarów związanych z bezpieczeństwem. Praca ta również zwięźle prezentuje stan sztuki oraz zarówno zalety, jak i wady istniejących rozwiązań używanych na całym świecie.

Wynikiem tej pracy jest koncepcja wieloplatformowego frameworku, który pozwala w prosty sposób skonfigurować obszary związane z zabezpieczeniami dla każdego rodzaju aplikacji mobilnej. Co więcej, framework ten gwarantuje łatwe modyfikacje oraz rozbudowę domyślnych implementacji. Jako część tej pracy zrealizowano odpowiedni prototyp. W celu udowodnienia poprawności jego działania, powstała również przykładowa aplikacja mobilna.

Słowa kluczowe: Informatyka, aplikacje mobilne, bezpieczeństwo, wieloplatformowość.

Table of Contents

1 Introduction

This chapter describes in detail goals of the thesis and motives which stand behind it. It also briefly summarises a proposed solution and results that were achieved.

1.1 Goals

The aim of this thesis is to analyse existing software and protocols used for securing mobile applications and propose a new framework which meets standards and is easy to use by developers. Such framework should guarantee best possible security and simultaneously be straightforward in usage to both junior and senior developers. It should allow junior developers setting up secure applications from scratch with minimal effort while senior developers can additionally intervene in the framework logic by overriding particular parts of it. To fulfil these requirements, the framework must be divided into small components and guarantee that each one could be overridden by custom implementation without affecting logic as a whole. It also must be cross-platform and datastore independent to allow targeting multiple operating systems and platforms.

1.2 Motives

The author's motivation to take up the matter being the subject of this thesis relates to the fact that he had spent a lot of time on understanding and setting up security in commercial and non-commercial projects targeting many business areas. Thoughts about available frameworks and protocols led the author to decision to propose and build something new which could facilitate easier development for those ones working with security.

1.3 Solution concept

The proposed solution is a framework which gathers best concepts from existing libraries and frameworks to provide a simple, extensible and secure way to develop mobile applications with server side API based infrastructure. The solution consists of coexisting frameworks targeting three areas:

- Identity server – a framework used for setting up identity server, which is an API responsible for user identity and access management.
- Resource server – a framework used for setting up resource server, which is an API representing access layer to protected resources.
- Mobile application – a framework used for setting up mobile application to communicate securely with Identity and Resource Server.

All the mentioned frameworks are based on shared SDK which guarantees that each of them uses common models and interfaces. Each component has been built in a way that introduces significant simplifications in the process of secure software development. In exemplary use case one will simply have to call a method named 'SignIn' instead of performing several activities like writing code responsible for processing OAuth 2.0 grants, obtaining information about returned tokens and saving them in a secure way. As it can be seen, the framework enables usage of comprehensive solution which – to say colloquially - will take care of everything.

Other important thing is the fact that the framework is a configurable, plug-in based solution which allows overriding very atomic components, like logic responsible for storing token data on mobile device or whole areas like server side datastore access logic. Datastore access layer is in fact a part which is one of the most important elements of the server side solution. It is a totally plug-in based component which implements unit of work and repository patterns. It has been created in such a way

that it is totally independent of category of used datastore. It can be anything - relational database, NoSQL database, API interface or even a flat file if a customer wants to use such kind of a persistent store. It also provides implementation of two plug-ins for SQL database engine and gives an ability to implement own plug-in, if required.

1.4 Prototype

The prototype consists of three parts: identity server, resource server and mobile application. Each of them is meant to present simplicity of the proposed framework usage. Mobile application is the most expanded part of the prototype as it required some effort to build user interface which would show practical and business use case, not only its technical aspects. As the core functionality of the mobile application, the author chose simple TODO list as it allows presenting briefly all CRUD operations which represent the most common use cases in all business applications.

2 Problem description

This chapter describes aimed problem and provides basic information required to understand it. It also presents areas which should be considered when talking about security in general and in case of mobile applications.

2.1 Understanding security

In general, software engineering is not the same activity when the security aspects need to be considered. All developers can write a code which properly handles business conditions, executes internal or external APIs and works efficiently. However, building secure software also requires thinking about things like how long and where to store information about user identity, how to manage user credentials, which encryption and decryption algorithms are best to be applied and how to eliminate all possible vulnerabilities. Gathering such knowledge requires from one a lot of time, as the number of standards, protocols and best practises are countless. For example, part of the specification for OpenID [3] protocol consists of almost 100 pages which have to be read and understood to say that one has skills to develop software which fulfils rules defined by this protocol. Additionally, the wealth of approaches and documentation is also an obstacle for good understanding of security areas. Without understanding it as a whole, a developer may make wrong choices during development because one standard used together with another one may cause potential problems which a developer might be not aware of.

2.2 Awareness of vulnerabilities

Besides taking care of security in general, it is also important to keep in mind that no matter how well secured software or framework has been built, there always exists a possibility to expose an application (and a user using it) for attacks or data loss. It can be done due to lack of awareness of potential vulnerabilities in the team of developers. For instance, even though an application is equipped with extraordinary security infrastructure for communication with API via internet and guarantees secure datastore at a device level, someone may temporary save user's credentials in a kind of unsecure data store which will enable stealing such information. Therefore, it is essential that a potential candidate for framework should address and identify as many vulnerabilities as possible in order to secure them. Additionally, developers must be instructed as well how to avoid mistakes which can cause problems not fully resolved by the framework.

2.3 Security in mobile applications

Applications targeting mobile devices require managing security in areas which are common with other platforms as well as things which are only specific to them. It is required as in case of mobile devices application runs on a user device hence a developer has no control over it. In such situation there are additional aspects to manage during a software development process. To fully understand challenges behind sentence 'security in mobile applications' one should get familiarised with some important facets which are briefly discussed in this chapter.

2.3.1 HTTPS and HSTS

One of the most obvious and the most important matters about which each developer should remember in a development process of any application is HTTPS. Recent years have shown a growing number of web applications using this protocol for communication purposes which is actually a good trend. Despite this fact, using HTTPS is not everything. Additional aspect which should always be taken into account is not only using secure communication but also ensuring that it is always used and no protected resources are exposed via HTTP protocol. There are multiple techniques applied to achieve it,

but the most known is HSTS which stands for HTTP Strict Transport Security. This mechanism is a kind of a guard assuring that each communication between client and server is executed using HTTPS even in case of an attempt to use HTTP.

2.3.2 Certificate pinning

Certificate pinning in a nutshell is 'the process of associating a host with their expected X509 certificate or public key' [4]. In other words, pinning gives a software an additional proof that a host used for communication is the one which should be trusted. Without certificate pinning each application communication with a server is exposed to potential men in the middle attacks.

MITM (man in the middle attack) [27] is a cryptologic attack which is an attempt to intercept messages dispatched during communication between two sites – client and server. For example, if site A (client) sends message to site B (server) 'the man in the middle' tries to impersonate site B by presenting own, mirrored API under the same IP address with difference in public key of the certificate.

As described in the definition, the MITM attack is very simple to perform for an attacker with appropriate abilities. Thanks to certificate pinning this danger is practically eliminated. In the author's opinion, this functionality should be a standard in the HTTPS communication use cases.

2.3.3 OpenID Connect

In the process of application development, one should always consider two security keywords which usually are used together. Many people also use them interchangeably what is a grievous mistake, thereby one should know the difference between them:

- Authentication (Authn[1]) – is a process of establishing information about user identity. In other words, the result of authentication process is information whether a particular user is the one who they intend to be.
- Authorization (Authz[2]) – is a process of establishing information whether a user has access to a particular resource, for example a list of contacts in e-mail.

At the beginning of XXI century a variety of companies developed a great number of approaches to handle these two areas properly and securely. The problem was that they rarely thought about the fact that users didn't want to authenticate to applications frequently and using different methods. It didn't take a long time when development community started to work on alternative approaches. One of them and currently the most widespread security protocol is OpenID Connect [2]. This framework describes in details basic rules which should be met to create applications in a secure and interoperable way. What is very important, it covers both areas authentication and authorization, as the solution is built on top of the Oauth 2.0 [3] protocol which is purely an authorization framework. It is also the first framework which is a 'mobile friendly' solution to address security problems as previous generations of OpenID Connect and other frameworks usually targeted only web applications.

As it is described in RFCs of OpenID Connect [2] and Oauth2.0 [3] frameworks, the core concept of the whole approach is to reduce the number of moments when a user has to enter their credentials in any application to the required minimum. This goal is the most important part of the whole initiative as user's credentials once intercepted may be easily used to act as this user and seriously impair their protected resources like for example e-mail box. In case of such a loss, the only way to revoke access to user data is to change credentials what sometimes can be difficult as an attacker may simply change user password and make it impossible. To facilitate this case, OpenID Connect framework and

[1] Shortcut for Authentication
[2] Shortcut for Authorization

its prerequisites introduced the concept of access tokens which are currently the basic entity used to access protected resources worldwide.

So, what is a token? In short, it is a string which is an official permission given to the application by a user to access their data. Such token usually has no reference to user identity and has limited lifetime. As such tokens are temporary, an attacker taking over access token will not have a chance to use it with no limits. Additionally, access tokens can consider only particular scopes of user data so getting a token does not necessarily mean an access to everything but only to limited range of user data, user photos or videos for instance. For better understanding we can compare access token to a hotel key card as usually does Aaron Parecki [5] during his presentations. As he says, the room door doesn't concern whether the person using the key card is the one authorised to use it, door only matter whether this particular key card has access to it at the particular moment of time or not. The responsibility for such concerns lies in hands of a receptionist who exchanges the hotel customer id card (in case of an application - the password) for a key card (in case of an application - the access token).

So, what happens when such access token expires? There are two options – user may re-enter credentials or use another token issued together with the access one. This additional token is named refresh token and allows issuing a new access token and its refreshing without asking a user to enter credentials again. Of course, a decision whether a refresh token should be in place or not depends on a particular use case and a few protocol and implementation restrictions.

2.3.4 Persistent store

Each security framework should consider that artefacts used in each authentication or authorization process should be persisted somehow. In case of mobile applications this activity is especially difficult because no matter what kind of store will be used it is beyond control of a developer. One can distinguish a few persistent stores which can be used depending on a particular use case:

- In memory – simple in memory store is the easiest and fastest option to save both confidential and public data but it does not fit well when one wants to use data after application closure.
- Preferences – simple key-value persistent store which gives an option to persist data durably. This approach is not recommended for data requiring protection as the preferences could be easily intercepted.
- Secure Storage – similar to Preferences but with a guarantee of data encryption. Access to it is protected using a key which is only available when a device is unlocked. This approach is recommended option for data requiring protection, but potential background processes can't use it unless a device is unlocked.
- SQLite database - simple and powerful database engine used worldwide. SQLite databases are popular and cross-platform option for data persistence. As the database, this type is in fact a file kept at a device level so that it is very simple to be intercepted in case of a device loss.
- Encrypted SQLite database – a very powerful extension for SQLite databases is encryption of a database file. In this case, the only option to access such database is to open it using a secret key. The only thing which should be deliberated is a secret key structure and the fact who should possess it or how to obtain it.

As it can be seen, there are a few common options which can be used for persistence of secure data at a device level. Every of them has both advantages and disadvantages and choice between them should be aligned to a particular use case.

2.3.5 Time-based One-time Password (TOTP)

The common standard in case of mobile applications is usage of OTPs - one time passwords. One time password can be met in two factor authentication scenarios, where such secret code is sent to a user using preferred channel (SMS or e-mail message) and is used together with the credentials as an supplementary factor to provide additional proof of user identity. Such code usually has fixed lifetime and is valid only for several minutes. So, what differs standard OTP from TOTP? As the abstract from RFC6238 states, TOTP is [6]:

'A time-based variant of the OTP algorithm provides short-lived OTP values, which are desirable for enhanced security.'

Basing on this statement, this kind of a code used to provide additional authentication factor to secure back channel communication between the client and the server. Important requirement for TOTP usage is the fact that both the client and the server must possess a special shared secret which is used for generation and validation of the generated TOTPs.

In the author's opinion, using TOTPs in back channel communication is a big enhancement in the area of secure communication over Internet and its usage in mobile applications development is very important.

2.3.6 Code obfuscation

Artefacts (e.g. files, settings) of mobile application are kept at a device level, thus it is very easy to perform a reverse engineering on it. One can simply connect to the device, download application libraries and reconstruct source code from them. The knowledge about application sources gives an option to find vulnerabilities even in well secured applications. The solution for this is a process of code obfuscation. In a nutshell, obfuscate code means rewriting it in a way that changes a code into a totally meaningless mixture of single characters. The aim of this process is to hamper the code reconstruction and understanding it to the maximum. A very good description of the obfuscation process as a whole and a detailed approach as well can be found in the book 'Obfuscation, A User's Guide for Privacy and Protest' by Finn Brunton and Helen Nissenbaum [28].

3 State of the art

It is obvious that developers community is the booming environment thus it is not hard to find ready solutions for all mentioned facets. One can easily find plenty of libraries and frameworks which facilitate process of securing all kind of applications, including mobile ones. Among all of them one can distinguish helper libraries like Identity Model [7], frameworks like Identity Server 4 [8], box solutions like Keycloak [9] or these delivered in SaaS model like Auth0 [10]. Such products usually facilitate understanding and using security protocols in development and enterprise scale but always leave some part of work in the hands of a developer. As the number of available solutions is significant, the state of art presents only these solutions which deserve special attention from the author's perspective.

3.1 Identity Model

Identity Model [7] is a set of certified libraries which facilitate common and repeatable development activities like:

- Communication with identity provider service in areas described by OpenID Connect and OAuth 2.0 protocols.
- JWT tokens handling.
- Encryption and decryption processes.
- Access tokens introspection.
- Scopes validation.

All these areas are essential and this library set is a good choice for one who would like to work with authentication and/or authorization in any kind of applications and build logic from scratch. Nevertheless, usage of these libraries requires some level of familiarity with security protocols being basis for this framework. In case of developers who want to just use such logic without digging into details, it is quite good but not the best choice.

3.2 IdentityServer4

IdentityServer4 [8] is a free ASP.NET Core framework which provides logic consistent with OpenID Connect and OAuth 2.0 protocols. It is an example of a mature framework which was initially build using ASP.NET Framework under name IdentityServer3. Both the current and the previous versions are certified OpenID Connect and OAuth 2.0 implementations. As IdentityServer is a framework not a library, it provides out of the box almost everything required to set up custom identity server from the beginning. One has only to set up Web API project, install framework from nuget package, implement a few interfaces and run an application. Basically, one day is enough to set up fully functional Identity Provider Web API, perform some customisations and deploy it, for example to the Azure App Service.

Possibility of customisations is probably the most important feature of IdentityServer. The whole framework is built using plug-in architecture with default implementations for almost every part of it. Places where implementation must be developed – for example logic responsible for user data retrieval – have also default test implementations which allow a developer a simple set-up and API start without digging into its details. This whole solution also allows simple turning on or turning off some functionalities based on configuration. From the author's perspective, all these facts are big advantages of the framework and he can recommend it for developers who want to build custom identity provider without wasting time for building things which are already available.

Notwithstanding the advantages, the IdentityServer framework still requires a deeper knowledge to use it consciously. Using default implementations and default settings is correct approach but usually one has to perform some modifications to align logic to particular business use case. Such activity requires performing changes in areas related to OpenID Connect client configuration which may introduce some potential problems, if done incorrectly.

To sum up, the whole framework is a very good and flexible solution but if one doesn't want to engage in implementation details which IdentityServer team left in hands of developers, they may want to choose out of the box SaaS or PaaS model. Other important factor which should be taken into account when choosing between a framework and complete software are funds and support.

3.3 Keycloak

The Keycloak [9], [11] is a mature and open source product from Red Hat which was designed to allow simply and freely using Identity and Access management logic out of the box. The whole solution is also a Certified OpenID Connect, OAuth 2.0 and SAML implementation. It provides a few features as well which are very important nowadays like:

- Single Sign-On.
- Social login.
- 2-factor authentication.
- LDAP integration.

To set up and start the Keycloak server as well as to add a few test users one requires at most 15 minutes. After that any simple client application which met Authn and Authz protocols can communicate with the server to obtain a first token. Apart from the simplicity, the whole solution is very configurable making it a very good option to choose when talking about security.

Nonetheless, no matter how this solution is flexible and easy to be applied, it still has some drawbacks. Being an open source product, it provides no official service support in case of any defects or updates of the software. This fact implies that such free solution is not a good choice for companies which build IT infrastructure based on external providers software delivery. In such case, there is usually no in house team available to maintain and support such software or a team has not enough competences to do it. To guarantee support, one has to move to paid option which is Red Hat Single Sign On. This product is built on top of Keycloak so it guarantees all its features.

3.4 WSO2 Identity Server

The Keycloak is of course not the only product of such a kind. One can find many similar solutions out of which WSO2 Identity Server [12] deserves mentioning. It is also an open source and provides basically all the same features as the Keycloak. What is worth highlighting is the fact that WSO2 provides identity provider software in two models:

- As a standalone application which can simply be downloaded and installed on any machine. This option is free but likewise the Keycloak support requires paid subscription.
- As a cloud solution which requires an account in WSO2 Cloud. This option requires a paid subscription based on a number of active users.

3.5 Auth0 service

The cloud model of software distribution is becoming increasingly bigger nowadays. In the author's opinion, the best cloud based SaaS option is to choose Auth0 [10]. This product is based on a

paid subscription which is, as in the case of WSO2, based on a number of active users. As the company claim on their official website, they provide ‘Authentication as a service’ and it is truly a fact. A lot of known companies already trusted Auth0. Among them we can mention for example Siemens, Mazda, Atlassian, Nvidia and many more. Abilities of this platform are very huge an it is very easy to set up a new application definition in just a few steps thanks to the intuitive user interface. As always, there is a question of the product price and a decision whether the expected product benefits exceed the costs to be carried. Naturally, if the company serves more than 100 000 users it should have funds to pay for such service but as always it depends on the use case.

4 Proposed solution

This chapter describes in detail a proposed solution. It presents core design concepts, points out strategic functionalities and requirements.

4.1 Course of action

According to analysis covered in previous chapters, the market is full of ready solutions which provide almost all functionalities required in common scenarios related to secure communication over internet connection between mobile client and server. On the other hand, one can find numerous frameworks which facilitate development of software which provides conceptually the same features as ready solutions. Choice between options is usually related to particular customer requirements and a few other factors:

- In case of box and SaaS solutions the first and foremost matter are funds. Majority of available products are available in a free option but only with limited functionality and without official support. What is also worth mentioning, so-called free tier is usually enough only for test and research purposes. The choice between box and cloud solution is also important. In case of software installed on own infrastructure potential costs are only related to support and updates of related issues. For SaaS model costs of subscription usually grow with number of users therefore making a decision to use it should be wise and preceded by solid analysis of costs and forecasts considering number of potential users and login sessions.
- Building own solution using either in house or vendor workforce requires knowledge and significant effort put into solution design and development. Process of such software delivery can be of course reduced by using available frameworks however it is still relatively big in comparison to standalone and SaaS products. On the flip side, building own software gives and option to ensure that it meets all requirements – common and specific ones which are not covered by the standard.

Taking all these aspects into consideration, the author came to the conclusion that building custom Authn and Authz solutions is a field at which proposed solution may introduce enhancements and new features in comparison to frameworks which are currently available.

4.2 Core concepts

In view of current state of the art, the most important facet which should be considered above all others is a consistency with OpenID Connect [2] and OAuth 2.0 [3] protocols. These protocols are currently a standard which is commonly used all over the world. Such approach guarantees that potential software is:

- Secure,
- Modern,
- Extensible,
- Cross-platform.

The second important factor is to build a software in a way which guarantees extensibility. Extensible means that individual components of the framework allow being replaced by custom implementation. To improve extensibility potential, it is also important to divide a whole software into components which are as small as it is possible. Thanks to such approach, prospective software enhancements will be easier and faster.

The third goal is to make the framework suitable for both experienced and junior developers so the solution development should be done bearing in mind KISS[3] principle. This assumption covers all areas of the created framework from pure implementation problems to naming of the classes and methods. As the solution is also meant to be extensible, which always introduces some level of complexity and abstraction, it is required to find a balance between simplifying a code and keeping it flexible.

4.3 Core functionalities

The final product being a result of this thesis should be a framework consisting of three packages targeting different logical components:

- Identity server Web API framework.
- Resource server Web API framework.
- Mobile application framework.

Each package should be a separated being built on top of shared SDK what guarantees that each functionality works using a common codebase. Figure 1 present conceptual framework relations with 'Common SDK' as a base for other frameworks.

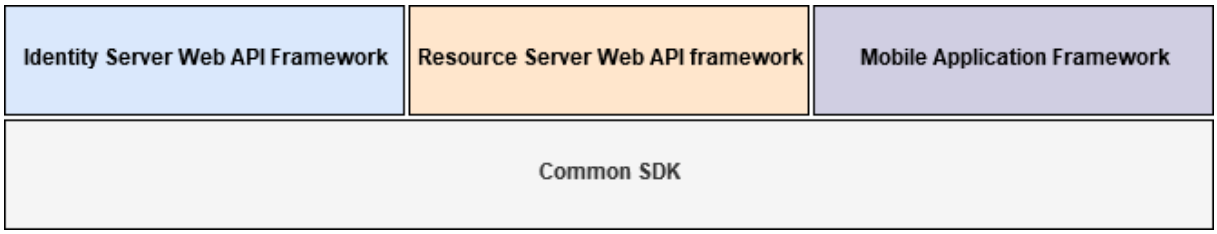


Figure 1: Proposed solution architecture concept. Own elaboration

4.3.1 Identity Server Web API framework

This package should deliver a framework which adds a support for Authn and Authz to the solution. In basic use case a developer has to:

- Initialize new, blank Web API project.
- Install packages with the framework.
- Implement a few basic interfaces (e.g. user store) or use test in-memory implementations.
- Add the framework to the pipeline.
- Run application and test it using rest client of choice.

Framework should guarantee consistency with protocols and standards by serving core OAuth 2.0 and OpenID Connect features with a few enhancements:

- New oauth grant type based on user PIN code and TOTP mechanism.
- Logic responsible for registration of the fact that user installed application at both: a device and the server level.
- Extension of Identity Server discovery response – serving additional JWKS (Json Web Key Set) entries.

[3] KISS stands for 'Keep it Simple, Stupid'. This design principle essence is to build software in a way which is as simple as possible. Such approach allows less experienced developers understanding a code.

The details of the implementation have been described in chapter 7 ‘MMAS server frameworks’.

4.3.2 Resource Server Web API framework

This package should deliver a framework which adds a support for securing protected resources by validation of access tokens passed together with HTTP requests. In basic use case a developer has to:

- Initialize new, blank Web API project.
- Install packages with the framework.
- Add framework to the pipeline.
- Add resource which should be protected.
- Run application and test it using rest client of choice.

The framework should guarantee that each resource exposed by the API is properly secured what means that it can be accessed only when the valid access token with proper scopes is provided within the HTTP request.

The details of the implementation have been described in chapter 7 ‘MMAS server frameworks’.

4.3.3 Mobile Application framework

This package should deliver a framework which adds a support for secure communication with the Identity Server and Resource server. Usage of the framework will require from a potential developer aligning coding approach to follow the MVVM design pattern principles. Additional requirement will be usage of the IoC container. The framework should deliver out of the box functionalities like:

- Application bootstrap (initialisation) during each start-up, including:
    - Identity Server discovery document retrieval [2].
    - Identity Server JWKS document retrieval [2].
    - Encrypted database(s) set-up.
- Sign-in with password support including:
    - Flow with password credentials grant [3].
    - Flow with authorization code grant [3].
- Sign-in with PIN code support.
- Certificate pinning support.

The details of the implementation have been described in chapter 8 ‘MMAS mobile framework’.

5 Technologies and tools

This chapter depicts technologies and tools used by the author during the process of solution implementation.

5.1 .NET

.NET (pronounced as ‘dot net’) is a programming platform built by Microsoft. The beginnings of .NET were seen at the end of the XX century. Throughout the years many versions of this framework have been released, starting from the early .NET Framework 1.0 up to .NET Core 3.0 which is about to be released.

The important milestone was a transformation from .NET Framework to .NET Core. The latter one can be executed on Windows, macOS and Linux based environments while .NET Framework can be run only on machines with Windows OS. This is possible thanks to the .NET Standard which - in a nutshell - is a collection of .NET APIs that have to be implemented for each platform. Figure 2 presents general .NET architecture.

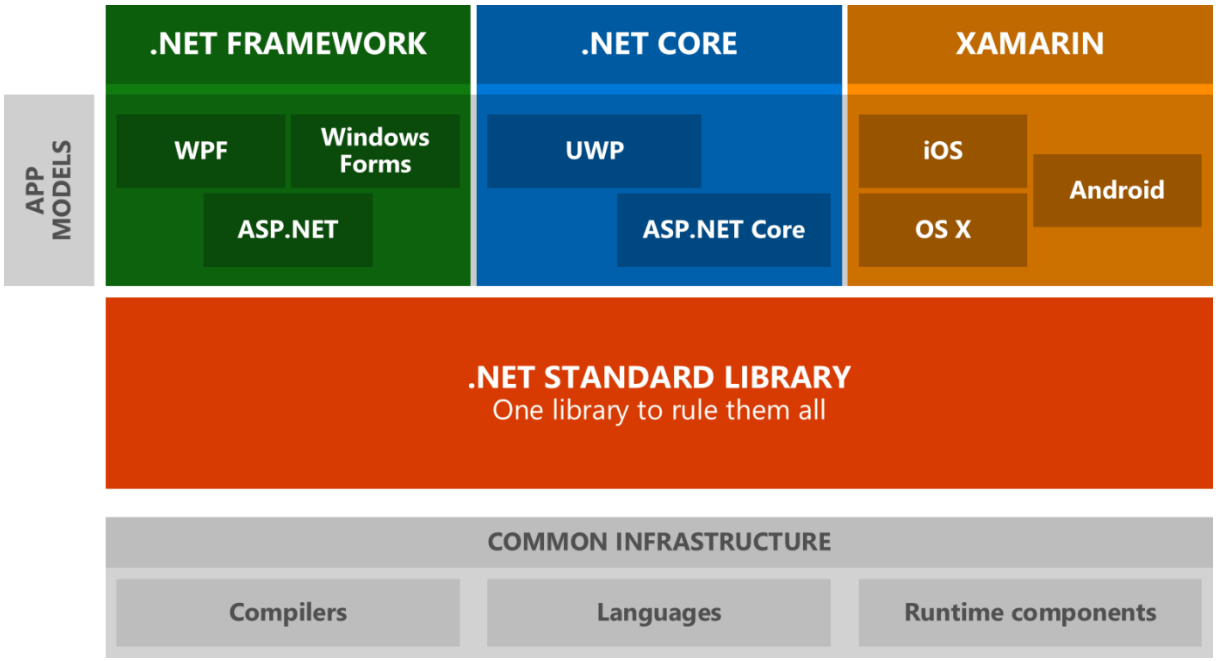


Figure 2: .NET architecture. Source: [14]

Currently .NET is a free and open source development platform which allows building cross-platform applications targeting multiple areas like web, mobile, cloud, desktop, gaming, machine learning and IoT.

.NET Standard can be understood as a common, fundamental basis for other .NET based frameworks. To put it briefly, it is a collection of .NET APIs which have to be implemented for each platform level. The fact that each platform uses shared environment allows developers to build one, transparent code which can be run on each platform using dedicated implementation of APIs under the hood. This framework has been used during the development of the framework libraries to ensure that it can target various .NET based platforms now and in the future.

.NET Core is a cross-platform framework being a part of .NET platform which enables building efficient and cross-platform applications targeting web, desktop (since .NET Core 3.0) and Universal

Windows Platform. In comparison to its precursor ASP.NET Framework, it is notably more performant and can target multiple operating systems. This framework is also much more friendly when working with cloud as it was designed during the time when cloud based applications popularity grew. It has been used to build Identity Server and Resource Server implementation prototype in version 2.2.

5.2 Xamarin Forms

Xamarin Forms is a cross-platform framework which enables building native applications for Android, iOS and Universal Windows Platform using .NET. This framework essence is to build one code which will be reflected on each platform with respective UI. In basic use case it allows sharpening up to 90% of the code. The amount of the platform specific code may change depending on the requirements for the UI but even in case of a sophisticated specification it still allows sharing approximately three quarters of the code. What is important, the framework is built on top of the .NET Standard thus the .NET Standard libraries can be easily used during the implementation. It is a powerful solution because one library written in .NET Standard can be referenced in Xamarin Forms, .NET Core and .NET Framework based solution. This framework has been used to build the mobile application prototype within the enterprise standards [1].

5.3 C#

As the official Microsoft website says [14] C# (pronounced as 'C Sharp') is a 'simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers'. Next to languages like Java, Python and JavaScript, C# is one of the most popular programming languages used worldwide.

5.4 Visual Studio 2017 / 2019

Visual Studio is the most popular IDE[4] for software development on top of the .NET platform and not just only this one. This environment has been used during the process of implementation of the all components composing the framework. The lion's share of the work had been done in version 2017 but near the end of work, the author's decided to move work to version 2019 to ensure that implementation was being done using the latest available technology.

5.5 ReSharper

ReSharper is an extension for Visual Studio built by JetBrains company. This add-on has been released on a regular basis since the moment when version 2010 of this popular .NET IDE was released. This extension is a potent and modern option for developers who esteem opportunity to avoid common mistakes during code writing and application of intelligent IntelliSense[5].

5.6 Visual Studio Code

Visual Studio Code is a free and powerful code editor. Stackoverflow developer survey [15] has shown that it is the most popular IDE worldwide. Popularity of this tool has grown too such a level due to the following facts:

- It is free and distributed using MIT license.

[4] IDE stands for Integrated Development Environment.
[5] IntelliSense is a mechanism of autocompletion of code written in IDE. Basic example is a list of syntax specific suggestions which are available at a particular line of code.

- IDE has been built with extensibility principle and thanks to it developers community has delivered numerous extensions to it.
- Product is very light in comparison to other mature IDEs. Of course, for advanced scenarios it is still not enough but basic implementation can be done using this tool and only a handful of extensions.

5.7 REST Client

REST Client [16] is a significant extension for Visual Studio which allows sending HTTP based requests directly from Visual Studio code level. To use it, one must simply install extension, create file with .http or .rest extension and write a portion of straightforward code which represents a single HTTP request. The exemplary usage of the tool is presented on Listing 1.

Listing 1: Basic REST Client extension for Visual Studio Code usage

```
POST https://example.com/api/resources HTTP/1.2
Content-type: application/json
Authorization: Bearer token
X-Api-Key: Secret

{
    "name": "test",
    "description": "test"
}
```

This extension enables usage of all common HTTP features including headers, query parameters, path parameters and many more. Ability to use variables which can be set at project or user level is also a relevant advantage of this tool.

5.8 Microsoft SQL Server 2017

Microsoft SQL Server (MS SQL) [22] is a popular and efficient relational database engine and management system used worldwide. It is a core database related product of Microsoft team and is being actively developed. It is available in both commercial and free model under the Microsoft SQL Server Express name. Thanks to the student's license, the author had a chance to work on prototype implementation using Developer edition of this database engine.

5.9 Microsoft Azure services

Microsoft Azure is a set of cloud services which gives the opportunity to efficiently build and deploy applications with corresponding infrastructure in a straightforward way. Thanks to direct integration with Visual Studio IDE developers can deploy for example MVC based application to the cloud by usage of one button. As part of the prototype, exemplary Web APIs has been deployed and hosted using Azure App Services with Azure SQL database.

5.10 PlantUML

PlantUML [18] is a free and open source tool aiming at fast and efficient creation of standard UML diagrams like Sequence, Activity od Class one. It can be used via variety of available tools like:

- Command Line.

- Web Application.
- Extensions for Visual Studio Code.

What is also important, each diagram can be exported to many target formats including .png and .svg.

5.11 Entity Framework

Entity Framework is a mature and open source ORM (object-relational mapping) framework used across the globe. In short, it provides technology which allows building data-oriented applications in a model driven way. It delivers out of the box whole logic responsible for communication with the database with an option to use raw SQL statements as well. The framework has been actively developed since .NET Framework 3.5 release and soon an Entity Framework Core 3.0 is about to be released together with .NET Core 3.0. Current version of Entity Framework Core can be used with many database engines including MS SQL, MySql, Oracle, PostgreSQL and a few more. Depending on the chosen engine, there is a need to install dedicated nuget packages with an appropriate provider.

5.12 NHibernate

NHibernate is another popular ORM framework which derives from its Java prerequisite named Hibernate. Alike the Entity Framework, it enables easy build of data-oriented software in a model driven way. It also supports several database engines in many versions.

5.13 SQLite-net

SQLite-net [24] is an open source and very light .NET library used for working with SQLite based datastore. Thanks to its cross-platform architecture it provides an option to use it on many platforms thus it is widely used in Xamarin Forms based applications. From 2018 it is also available in the version which supports SQLCipher [25] extension for SQLite databases. Its main advantage is that databases can be encrypted in an efficient and easy to use method.

6 MMAS solution

This chapter describes details of the solution implementation. It briefly presents its architecture and presents dependencies between components being a part of the framework.

6.1 Architecture

The solution has been modelled in a way which ensures both, business and technical separation between projects composing it. The code map on Figure 3 presents relations between main projects and shows how they have been aggregated into logical groups. The key name **MMAS** used from now on is a shortcut for **M**odern **M**obile **A**pplications **S**ecurity.

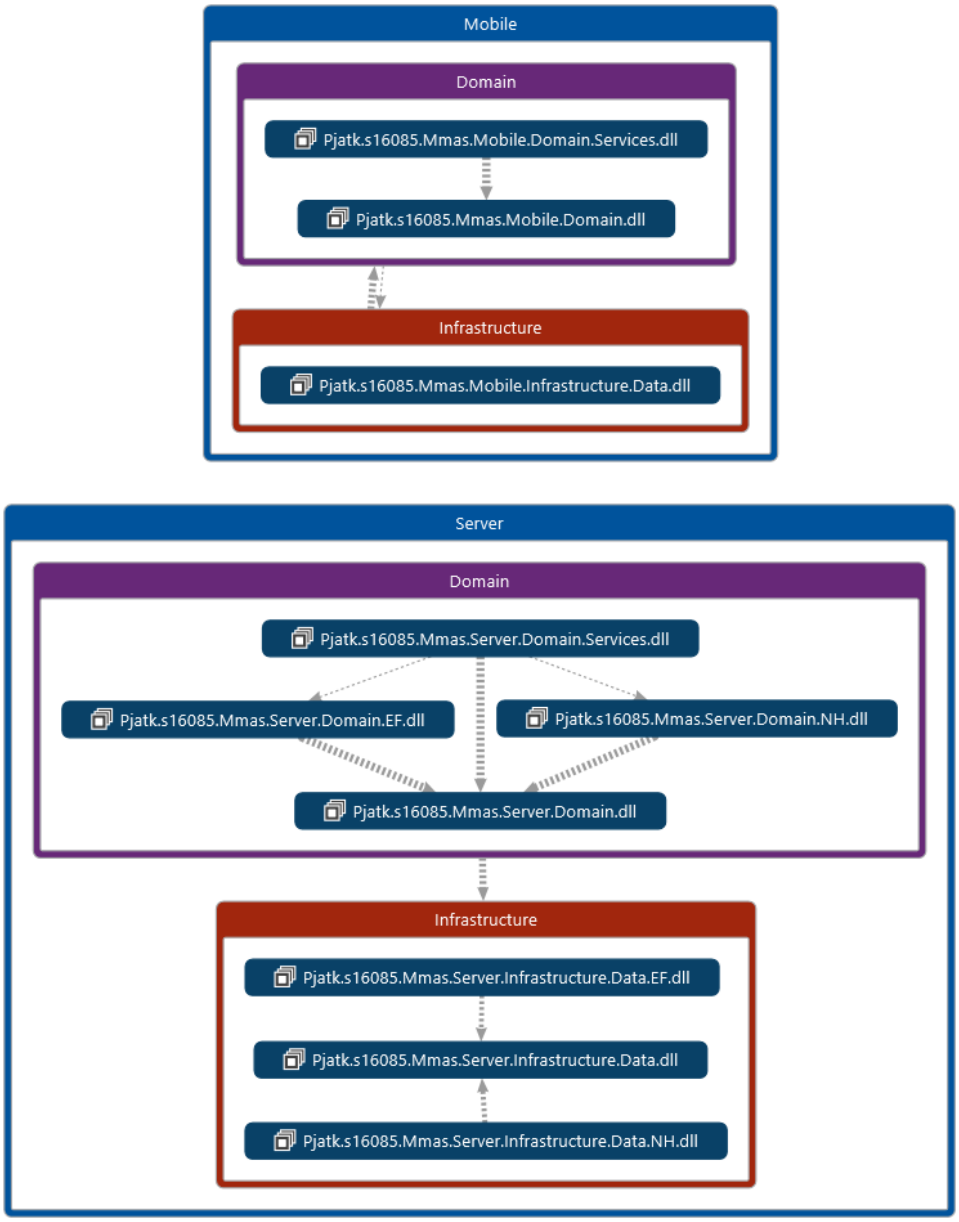


Figure 3: MMAS framework structure. Generated using [17]

One can distinguish the following projects with their responsibilities:

- **Common** – aggregates projects representing common models and logic. This project group has been omitted at the Figure 3 to simplify the code diagram.
    - **CrossCutting** – contains classes that provide models and logic which usage is repeatable across all components.
- **Server** – aggregates projects being a part of a server side framework.
    - **Domain**
        - **Server.Domain** – contains classes and interfaces which represent domain of the server side framework, e.g. datastore entities and services definitions.
        - **Server.Domain.NH** – contains classes which represent integration between domain and NHibernate framework.
        - **Server.Domain.EF** - contains classes which represent integration between domain and Entity Framework Core framework.
        - **Server.Domain.Services** – contains classes which represent implementation of server side framework business logic.
    - **Infrastructure**
        - **Server.Infrastructure.Data** – contains classes and interfaces which define generic layer used for communication with the datastore.
        - **Server.Infrastructure.Data.NH** – contains NHibernate specific classes and logic used for communication with the datastore.
        - **Server.Infrastructure.Data.EF** – contains Entity Framework Core specific classes and logic used for communication with the datastore.
- **Mobile** – aggregates projects being a part of mobile side framework.
    - **Domain**
        - **Mobile.Domain** - contains classes and interfaces which represent domain of the mobile side framework, e.g. datastore entities and services definitions.
        - **Mobile.Domain.Services** - contains classes which represent implementation of mobile side framework business logic.
    - **Infrastructure**
        - **Mobile.Infrastructure.Data**

6.2 Source control and deployment model

As the target of the implementation was not to build a complete framework with dedicated processes of continuous integration and deployment, the approach to this area has been simplified. Depending on the author's personal experience, the following areas have been managed as described below:

- For source control the free account at Atlassian Bitbucket was used with standard git repository.
- For deployment of exemplary application, there were used Microsoft Azure Portal App Services.
- The deployment process was executed manually using a dedicated Visual Studio integration layer.
- For deployment of exemplary Android mobile application direct integration between Visual Studio and the device was used. In case of iOS platform only a simulator was used as the author does not possess iOS device.

7 MMAS server frameworks

This chapter describes details of the identity and resource server part of the MMAS framework. It presents core components and depicts crucial parts of the implementation.

7.1 Generic datastore concept

To say that a framework is a cross-platform and generic solution it has to be datastore independent. To guarantee that MMAS server side framework will fulfil this requirement, two noted practises have been put into a place:

- Unit of Work design pattern which affirms that when multiple repositories related to the same datastore are in use they always share the same context / session / connection to it. The main purpose of such approach is a general support for transactions.
- Repository design pattern which ensures that all CRUD operations should be exposed using generic and standardised interface which hides its implementation details.

These two patterns are nothing new and usually each ORM library supports or delivers it in one way or another. Due to simplicity of its usage, developers often write a code which becomes ORM dependent and in case there is a demand for changes in the area of datastore, any amendment may become problematic. Such changes are very rare in case of common applications implementation as the datastore does not change during its lifetime. On the other hand, in case of generic frameworks like MMAS the decision how to communicate with datastore without knowledge about it is crucial. To safely and consciously manage this area, the author decided to build a set of generic interfaces and base implementations which should facilitate area of communication with a datastore, implement both patterns and allow framework integration with any kind of a datastore.

7.2 Generic datastore implementation

Among several datastore related beings, the basic and the most atomic one is an interface named *IDatabaseEntity* (Listing 1). In a word, it is a kind of a marker which is used to dynamically determine classes representing datastore entities. If a class derives from this interface MMAS framework will easily recognise it and try to put into mappings collection.

Listing 2: *IDatabaseEntity* interface definition

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.Interfaces
{
    public interface IDatabaseEntity
    {
    }
}
```

The next important element is abstract implementation of the *IDatabaseEntity* interface (Listing 3). It provides generic and virtual definition of an identifier of the entity. Its usage is helpful in scenario of retrieving single entity using its identifier without knowing anything about it.

Listing 3: Generic *DatabaseEntityWithId* implementation

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.Models
{
    public abstract class DatabaseEntityWithId<T> : IDatabaseEntity
```

```
    {
        public virtual T Id { get; set; }
    }
}
```

Abstract definition of *DatabaseEntityWithId* allows defining repository interface and base implementation of it, as it is presented on Listing 4 and Listing 5.

Listing 4: *IDatabaseRepository* interface definition

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.Interfaces
{
    public interface IDatabaseRepository
    {
        void Save<T>(T entityToSave) where T : class;

        IQueryable<T> Query<T>() where T : class;

        T GetById<T, TK>(TK id) where T : DatabaseEntityWithId<TK>;

        void Update<T>(T entityToUpdate) where T : class;

        void Delete<T>(T entityToDelete) where T : class;
    }
}
```

Listing 5: Generic *BaseDatabaseRepository* implementation

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.Repositories
{
       public abstract class BaseDatabaseRepository : IDatabaseRepository
       {
              public void Save<T>(T entityToSave)
                     where T : class
              {
                     // Place for generic logic related to save operation
                     // ...
                     this.SaveInternal<T>(entityToSave);
              }

              public abstract void SaveInternal<T>(T entityToSave)
                     where T : class;

              public IQueryable<T> Query<T>() where T : class
              {
                     // Place for generic logic related to query operation
                     // ...
                     return this.QueryInternal<T>();
              }

              public abstract IQueryable<T> QueryInternal<T>()
                     where T : class;

              public T GetById<T, TK>(TK id)
                     where T : DatabaseEntityWithId<TK>
              {
                     // Place for generic logic related to get by id operation
```

```
                // ...
                return this.GetByIdInternal<T, TK>(id);
        }

        public abstract T GetByIdInternal<T, TK>(TK id)
                where T : DatabaseEntityWithId<TK>;

        public void Update<T>(T entityToUpdate)
                where T : class
        {
                // Place for generic logic related to update operation
                // ...
                this.UpdateInternal<T>(entityToUpdate);
        }

        public abstract void UpdateInternal<T>(T entityToUpdate)
                where T : class;

        public void Delete<T>(T entityToDelete)
                where T : class
        {
                // Place for generic logic related to delete operation
                // ...
                this.DeleteInternal<T>(entityToDelete);
        }

        public abstract void DeleteInternal<T>(T entityToDelete)
                where T : class;
    }
}
```

Such approach assumes existence of a base, abstract repository which is a thin wrapper between interfaces seen by business logic layer and the real, ORM specific implementation. As such, it gives an option to intercept a moment between the call to for example a *Save* or *Update* method of the repository and the real persistence of the changes to datastore. The following example presents basic usage of this opportunity which is an update of date and time tracking properties of the entity being subject to changes.

Listing 6: Generic *BaseDatabaseRepository* implementation usage

```
public void Save<T>(T entityToSave) where T : class
{
    DateTime utcNow = TimeHelper.NowUtc;

    if (entityToSave is ITimeStampEntity timeStampEntityRepresentation)
    {
        timeStampEntityRepresentation.TimeStamp = utcNow;
    }

    if (entityToSave is IEntityWithCreationDatetime entityWithCreationDatetime)
    {
        entityWithCreationDatetime.CreatedAtUtc = utcNow;
    }

    if (entityToSave is IEntityWitLastUpdateDatetime entityWitLastUpdateDatetime)
    {
        entityWitLastUpdateDatetime.UpdatedAtUtc = utcNow;
```

```
    }

    this.SaveInternal<T>(entityToSave);
}
```

The only thing which remains to complete implementation of the ORM specific plug-in is to write a little portion of integration code what is briefly presented on Listing 7 and Listing 8.

Listing 7: Entity Framework Core *BaseDatabaseRepository* implementation fragment

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.EF
{
    public class EntityFrameworkRepository : BaseDatabaseRepository
    {
        private readonly DbContext _dbContext;

        public EntityFrameworkRepository(DbContext dbContext)
        {
            this._dbContext = dbContext;
        }

        public override void SaveInternal<T>(T entityToSave)
        {
            DbSet<T> dbSet = this._dbContext.Set<T>();

            dbSet.Add(entityToSave);

            this._dbContext.SaveChanges();
        }
    }
}
```

Listing 8: NHibernate *BaseDatabaseRepository* implementation fragment

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.NH
{
    public class NHibernateRepository : BaseDatabaseRepository
    {
        private readonly ISession _session;

        public NHibernateRepository(ISession session)
        {
            this._session = session;
        }

        public override void SaveInternal<T>(T entityToSave)
        {
            this._session.Save(entityToSave);
        }
    }
}
```

As it can be seen on presented examples the final, ORM specific implementations are dependent on either *DbContext* instance in case of Entity Framework or *ISession* instance in case of NHibernate. Both compose the implementation of unit of work design pattern as is. Such approach is not a complete solution because from business logic level there is no control over the unit of work as it is hidden in the repository. Such situation causes that in case of multiple CRUD operations executed one by one they will be executed against datastore in a way which is totally managed by ORM specific implementation. A solution for this problem is introduction of the simple, abstract unit of work definition which holds

the true implementation of it and is a source for the repository. It should guarantee support for transactions management as well.

Listing 9: *IDatabaseUnitOfWork* interface definition

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.Interfaces
{
    public interface IDatabaseUnitOfWork : IDisposable
    {
        void Flush();

        IDatabaseTransaction BeginTransaction();

        void Close();

        object GetBaseUnitOfWork();

        IDatabaseRepository Repository { get; }
    }
}
```

What is an additional and very important fact, the unit of work is a disposable being and it is important to keep it as long as it is required. This fact implies that single business logic execution may involve several usages of the unit of work. Furthermore, sometimes it is required to use multiple units of work related to different datastores. Another design pattern addresses this problem which is the factory.

Factory is a design pattern which assumes that responsibility for creating instances of some kind of an object is shifted onto class called factory.

To support multiple datastores, the author has designed *IDatabaseUnitOfWorkFactory* interface as it is presented on Listing 10.

Listing 10: *IDatabaseUnitOfWorkFactory* interface definition

```
namespace Pjatk.s16085.Mmas.Server.Infrastructure.Data.Interfaces
{
    public interface IDatabaseUnitOfWorkFactory
    {
        IDatabaseUnitOfWork GetUnitOfWork();

        IDatabaseUnitOfWork GetUnitOfWork(string name);

        void ValidateSchema();

        void ValidateSchema(string name);
    }
}
```

This one and the previous interface has to be implemented for each ORM used, but from business logic perspective it is transparent what is being done under the hood. For example, a call to factory method *GetUnitOfWork("CompanyDatastore1")* and *GetUnitOfWork("ComapnyDatastore1")* may retrieve a hidden instance of the unit of work which holds connection to a totally different datastores. Usually, the process of initialisation of connection to database is effortful. Bearing this in mind, the process of bootstrapping the unit of work factory was reflected in the code which is run during application start-up. Example presented on Listing 11 is a basic usage of the unit of work factory with simple CRUD operations and transaction management.

Listing 11: Unit of work exemplary usage

```
using (var unitOfWork = factory.GetUnitOfWork("Example"))
```

```
{
    using (var transaction = unitOfWork.BeginTransaction())
    {
        var something1 = new Something1()
        {
            // Setting data
        };

        unitOfWork.Repository.Save(something1);

        var something2 = new Something2()
        {
            // Setting data
        };

        unitOfWork.Repository.Save(something2);

        transaction.Commit();
    }
}
```

Approach presented on Listing 11 introduces additional work which has to be done to build logic aligned with proposed approach but after all it introduces general regulation for communication with datastore. It also leads to code simplification due to its clarity and transparency. MMAS server side framework has been implemented using approach presented in this chapter. What is more, the author believes that this infrastructure may be moved to separated library and become a standalone product used in many different implementations.

7.3 MMAS middleware

One of the author's principles was not to reinvent the wheel during the implementation, therefore the whole framework was built on top of the existing one - IdentityServer 4. Decision to pick such a direction of work was made due to two important facets:

- Chosen framework delivers OpenID Connect and OAuth 2.0 protocols implementation out of the box.
- As it states in the documentation '*IdentityServer uses the permissive Apache 2 license that allows building commercial products on top of it. It is also part of the .NET Foundation which provides governance and legal backing.*' [8]

Packages which have been used are:

- IdentityServer4 – it is a package which delivers middleware used for setting up pure identity server 4 functionalities in ASP.NET Core Web API.
- IdentityServer4.AccessTokenValidation – it is a package which delivers middleware used for setting up validation of access tokens in ASP.NET Core Web API.

This decision of course has consequences – dependency on the chosen framework. But in case of Identity Server 4, which is a brilliant example of extensible software, that did not have noticeable impact on the implementation process. On the other hand, one needs to ensure that default Identity Server 4 functionalities are not overridden. Therefore, it is crucial to safely extend it with new features without improper interfering into other ones.

The MMAS framework was built to allow using it like standard ASP.NET Core frameworks – as a middleware which can be added in Startup class of the application [19]. Therethrough, MMAS logic can be easily added to any project by application of simple extension methods as it is shown on Listing 12

and Listing 13. In order to improve the clarity of listings, the logging logic and properties declaration were removed.

Listing 12: MMAS Identity Server framework usage in Startup class

```
public class Startup
{
    public Startup(
       IConfiguration configuration,
       IHostingEnvironment hostingEnvironment,
       ILoggerFactory loggerFactory)
    {
        this.Configuration = configuration;
        this.HostingEnvironment = hostingEnvironment;
        this.LoggerFactory = loggerFactory;

        this._logger = this.LoggerFactory.CreateLogger<Startup>();
    }

    public void ConfigureServices(IServiceCollection services)
    {
        string applicationRootPath = AppDomain.CurrentDomain.BaseDirectory;

        // User store implementation
        services.AddScoped<IUserStore, TaskKitUserStore>();

        services.AddMmasIdentityServer(options =>
        {
            options.Caching.ClientStoreExpiration = new TimeSpan(0, 6, 0);
            options.Caching.CorsExpiration = new TimeSpan(0, 6, 0);
            options.Caching.ResourceStoreExpiration = new TimeSpan(0, 6, 0);
        })
        .AddMmasIdentityServerEntityFrameworkCoreSupport(
              typeof(EntityFrameworkTaskKitDatabaseMappingProvider))
        .PerformMmasIdentityServerInitialization(
              applicationRootPath, LogDesc);
    }

    public void Configure(IApplicationBuilder app)
    {
        if (this.HostingEnvironment.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseHsts();
        }
```

```
        app.UseHttpsRedirection();

        app.UseMvc();

        app.UseStaticFiles();

        app.UseMvcWithDefaultRoute();

        app.UseMmasIdentityServer();
    }
}
```

Listing 13: MMAS Resource Server framework usage in Startup class

```
public class Startup
{
    public Startup(
       IConfiguration configuration,
       IHostingEnvironment hostingEnvironment,
       ILoggerFactory loggerFactory)
    {
        this.Configuration = configuration;
        this.HostingEnvironment = hostingEnvironment;
        this.LoggerFactory = loggerFactory;

        this._logger = this.LoggerFactory.CreateLogger<Startup>();
    }

    public void ConfigureServices(IServiceCollection services)
    {
        string applicationRootPath = AppDomain.CurrentDomain.BaseDirectory;

        // User store implementation
        services.AddScoped<IUserStore, TaskKitUserStore>();

        services.AddMmasResourceServer(options =>
        {
            options.Authority = "https://mmas-auth.azurewebsites.net";
            options.ApiName = "api.taskkit";
            options.ApiSecret = "api.taskkit.secret";
            options.EnableCaching = true;
            options.CacheDuration = TimeSpan.FromMinutes(10);
        })
        .AddMmasResourceServerEntityFrameworkCoreSupport(
              typeof(EntityFrameworkTaskKitDatabaseMappingProvider),
              typeof(EntityFrameworkMmasDatabaseMappingProvider))
        .PerformMmasResourceServerInitialization(
```

```
                applicationRootPath, LogDesc);

            services.AddAuthorization(options =>
            {
                options.AddPolicy("TaskKitToDoItems", builder =>
                {
                    builder.RequireScope("api.taskkit.todoitems.read.own");
                    builder.RequireScope("api.taskkit.todoitems.create.own");
                    builder.RequireScope("api.taskkit.todoitems.patch.own");
                    builder.RequireScope("api.taskkit.todoitems.delete.own");
                });
            });
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (this.HostingEnvironment.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseAuthentication();

            app.UseHttpsRedirection();

            app.UseMvc();

            app.UseStaticFiles();

            app.UseMvcWithDefaultRoute();

            app.UseMmasResourceServer();
        }
    }
```

7.4 Dynamic logic routing

Default approach to add a new resource with logic in ASP.NET Core is realised by adding a new controller [20]. In case of the framework there is no option to do it hence it is required to add a new logic via middleware. Default approach is based on the map between HTTP path and logic which should be executed when HTTP request matches the path. Such software works but cannot be delivered in framework which is IoC driven.

To manage this, it is required to define a being which holds such map and dynamically chooses implementation to be executed. Such approach was applied with success by Identity Server 4 team and is a part of the framework. Despite this fact, the author decided to implement own version of this logic. This decision was driven by the justification that usage of Identity Server 4 approach for dynamic logic routing requires dependency on the whole framework. As MMAS resource server framework should be definitely lighter than identity server one, own implementation is far better than reliance on Identity Server 4 framework.

To achieve expected results, the following beings have been defined:

- *IMmasHttpActionHandler* – interface which defines class executing business logic as a response for HTTP request passed to it.
- *MmasHttpActionHandlerDefinition* – class which holds relation between path and *IMmasHttpActionHandler*.
- *IMmasHttpActionRouter* and *DefaultMmasHttpActionRouter* – interface and class implementing it which determines implementation of *IMmasHttpActionHandler* responsible for execution of logic associated with a particular path.
- *MmasResourceServerMiddleware* – middleware which is responsible for usage of above beings to handle HTTP requests in ASP.NET Core pipeline.

Practical usage example is shown on Listing 14, Listing 15, Listing 16 and Listing 17

Listing 14: *IMmasHttpActionRouter* interface definition

```
namespace Pjatk.s16085.Mmas.Server.Domain.Interfaces
{
    public interface IMmasHttpActionRouter
    {
        Task<IMmasHttpActionHandler> DetermineHandler(
              HttpContext httpContext);
    }
}
```

Listing 15: *IMmasHttpActionHandler* interface definition

```
namespace Pjatk.s16085.Mmas.Server.Domain.Interfaces
{
    public interface IMmasHttpActionHandler
    {
        Task ProcessAsync(HttpContext httpContext);
    }
}
```

Listing 16: *DefaultMmasHttpActionRouter* implementation

```
namespace Pjatk.s16085.Mmas.Server.Domain.Services
{
    public class DefaultMmasHttpActionRouter : IMmasHttpActionRouter
    {
        private readonly ILogger _logger;

        private readonly IEnumerable<MmasHttpActionHandlerDefinition> _handlersDefinitions;
```

```
        public DefaultMmasHttpActionRouter(
               ILoggerFactory loggerFactory,
               IEnumerable<MmasHttpActionHandlerDefinition> handlersDefinitions)
        {
            this._logger = loggerFactory.CreateLogger<DefaultMmasHttpActionRouter>();
            this._handlersDefinitions = handlersDefinitions;
        }

        public async Task<IMmasHttpActionHandler> DetermineHandler(HttpContext httpContext)
        {
            IMmasHttpActionHandler httpActionHandler = null;

            foreach (MmasHttpActionHandlerDefinition handlerDefinition in
              this._handlersDefinitions)
            {
                if (httpContext.Request.Path.Equals(
                      handlerDefinition.Path,
                      StringComparison.OrdinalIgnoreCase))
                {
                    httpActionHandler = (IMmasHttpActionHandler)
       httpContext.RequestServices.GetService(handlerDefinition.HandlerType);

                    break;
                }
            }

            return await Task.FromResult(httpActionHandler);
        }
    }
}
```

Listing 17: *MmasResourceServerMiddleware* implementation

```
namespace Pjatk.s16085.Mmas.Server.Domain.Services
{
    public class MmasResourceServerMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly ILogger _logger;

        public MmasResourceServerMiddleware(
               RequestDelegate next,
               ILogger<MmasResourceServerMiddleware> logger)
        {
            this._next = next;
            this._logger = logger;
        }
```

```
        public async Task Invoke(
              HttpContext httpContext,
              IMmasHttpActionRouter httpActionRouter)
        {
            try
            {
                IMmasHttpActionHandler httpActionHandler =
                                    await httpActionRouter.DetermineHandler(httpContext);

                if (httpActionHandler != null)
                {
                    await httpActionHandler.ProcessAsync(httpContext);

                    return;
                }
            }
            catch (Exception ex)
            {
                throw;
            }

            await _next(httpContext);
        }
    }
}
```

Infrastructure presented on Listing 17 enables adding middleware to ASP.NET Core pipeline and then injecting new handlers in any number. Such logic was used to add enrollment endpoint handler at a resource server level. In order to deliver it, a useful extension method was added to MMAS framework what is presented on Listing 18.

Listing 18: Adding MMAS Identity Server 4 HTTP action handler

```
mmasBuilder.AddMmasHttpActionHandler<DefaultMmasHttpActionHandler>(
"Mmas", "mmas/enrollment".EnsureLeadingSlash());
```

The *AddMmasHttpActionHandler* extension method simply adds a map between path /mmas/enrollment and *DefaultMmasHttpActionHandler class*. Therefore, *MmasResourceServerMiddleware* can locate this class when specified path is hit by HTTP request passed into ASP.NET Core pipeline.

7.5 Enrollment concept

The most important moment for the new sign-in method based on a PIN code is enrollment. By enrollment we understand a fact of installation and registration of an application on a user's device. This event is correlated with a few important things which have to be shared with the server side framework:

- Enrollment identifier – it is unique identifier of an application installation which is generated at a device level. On the later stage, this identifier is used in majority of MMAS framework logic.
- TOTP shared secret – it is unique, randomly generated secret used during sign-in process to generate TOTPs used as additional factor in sign-in process. It is also generated at a device level.
- User PIN code – alphanumeric code used by a user for sign-in process. It replaces strong password usage.

These three pieces of information are generated and set up on a user's device and it is required to provide a secure, efficient and simple way to share this data with the server side. MMAS framework introduces such possibility with an assumption that to perform enrollment a user must firstly sign in using a traditional password. As a result of sign-in process, the application using MMAS framework obtains an JWT access token which can be used to perform enrollment operation on protected resource. This assumption implies also a fact that MMAS framework supports only sign-in scenarios and assumes that the user registration process is implemented independently.

The MMAS server side framework defines four beings which are in use in enrollment and sign-in processes. Three of them are defined and implemented as a part of the MMAS framework and one of them is an abstract definition which has to be defined in application which uses the framework:

- *IUser* – this interface defines a class which holds basic information about a user. It also provides a method named *ListClaims* which should deliver all Oauth 2.0 claims associated with a user. This interface is specific for applications using MMAS framework therefore it needs to be implemented independently.
- *IUserEnrollment* – this interface defines a class which holds connection between information about enrollment of a particular application on a user's device and a user itself. This interface has its default implementation in framework.
- *IUserTotpSecret* – this interface defines a class which holds connection between enrollment and TOTP shared secret generated at a device level. This interface has its default implementation in framework.
- *IUserPinCode* – this interface defines a class which holds connection between enrollment and a PIN code set by a user at a device level. This interface has its default implementation in framework.

Table 1,

Table 2, Table 3 and Table 4 present details of each definition.

Table 1: *IUser* interface properties

Name	**.NET type**	**Description**
SubjectId	string	Unique user identifier. In OAuth 2.0 language it is known as the subject identifier
Username	string	Username set during registration process
EmailAddress	string	E-mail address set during registration process
PasswordHashed	string	Cryptographic hash of a user password
IsActive	bool	Boolean which determines whether a user is active or not
CreatedAtUtc	DateTime	Datetime at UTC time zone when user data has been initially created
UpdatedAtUtc	DateTime	Datetime at UTC time zone when user data has been recently updated

Table 2: *IUserEnrollment* interface properties

Name	**.NET type**	**Description**
SubjectId	string	Unique user identifier. In OAuth 2.0 language it is known as the subject identifier
EnrollmentId	string	Unique identifier of the application installation on a device
CreatedAtUtc	DateTime	Datetime at UTC time zone when user data has been initially created
UpdatedAtUtc	DateTime	Datetime at UTC time zone when user data has been recently updated
IsDeleted	bool	Boolean which determines whether enrollment is deleted or not
DeletedAtUtc	DateTime?	Datetime at UTC time zone when enrollment was deleted

Table 3: *IUserPinCode* interface properties

Name	**.NET type**	**Description**
EnrollmentId	string	Unique identifier of the application installation on a device
PinCodeHash	string	Cryptographic hash of a user PIN code
CreatedAtUtc	DateTime	Datetime at UTC time zone when user data has been initially created
UpdatedAtUtc	DateTime	Datetime at UTC time zone when user data has been recently updated
IsDeleted	bool	Boolean which determines whether enrollment is deleted or not
DeletedAtUtc	DateTime?	Datetime at UTC time zone when enrollment was deleted

Table 4: *IUserTotpSecret* interface properties

Name	**.NET type**	**Description**
EnrollmentId	string	Unique identifier of the application installation on a device
TotpSecret	string	Plain value of TOTP shared secret set during enrollment process
CreatedAtUtc	DateTime	Datetime at UTC time zone when user data has been initially created
UpdatedAtUtc	DateTime	Datetime at UTC time zone when user data has been recently updated
IsDeleted	bool	Boolean which determines whether enrollment is deleted or not
DeletedAtUtc	DateTime?	Datetime at UTC time zone when enrollment was deleted

Simplified process of obtaining those beings was presented on Figure 4.

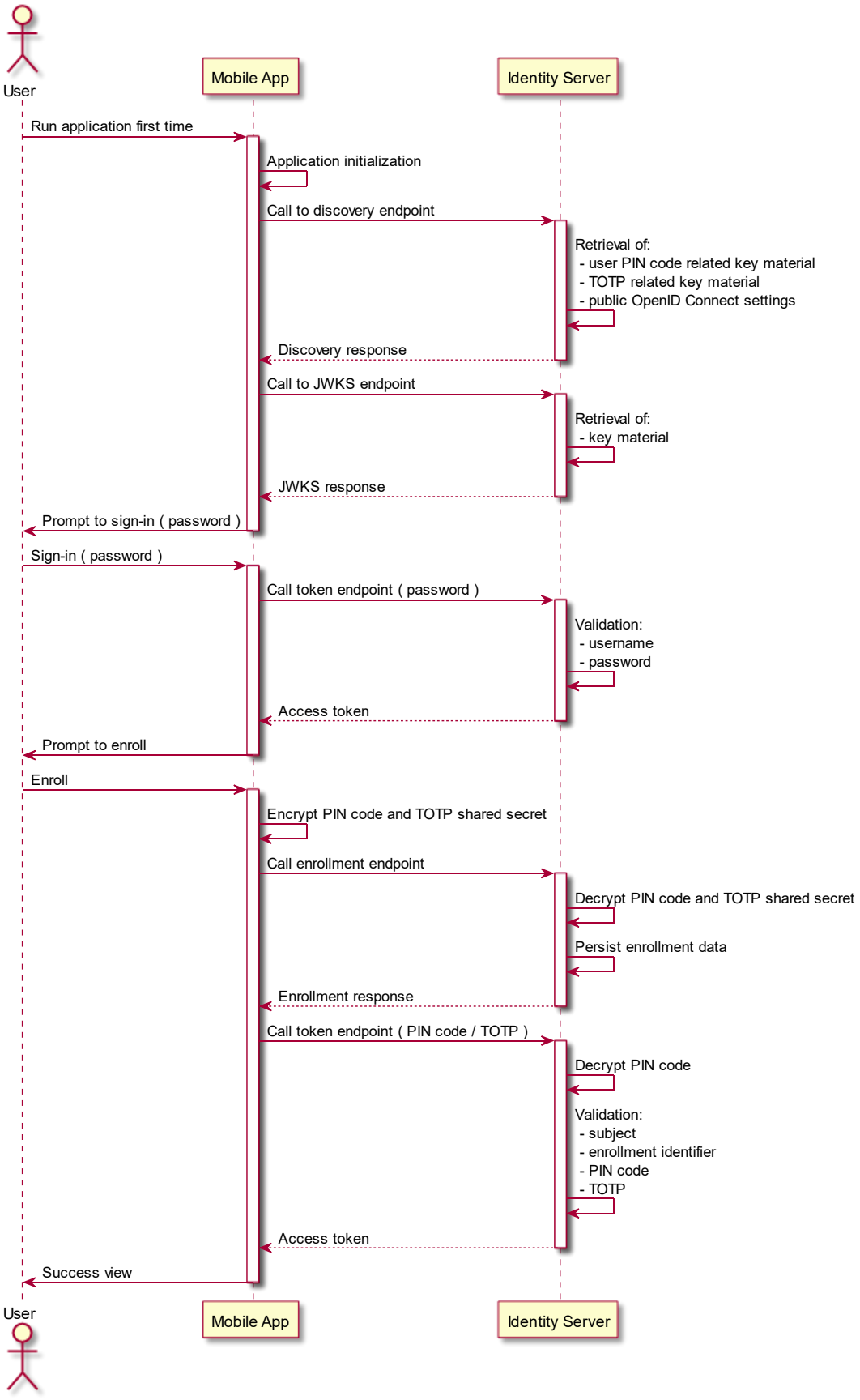


Figure 4: Enrollment process. Own elaboration

7.6 MMAS PIN Code grant

In the previous chapter the author has focused on a final operation which is enrollment. After the enrollment process there is performed a call for an access token using user's PIN code and TOTP generated. This functionality is a new logic added to default OpenID Connect middleware. It is possible thanks to Identity Server 4 feature – extension grant [21]. In short, it allows adding custom OAuth 2.0 grant implementation which is not delivered as part of the framework.

Request parameters which are required to call API are as follows:

- *grant_type* – informs the framework which grant should be used. Set to *mmas_pin_code_grant*.
- *client_id* – informs the framework about an identifier of client which executes grant.
- *client_secret* – informs the framework about a secret (password) of client which executes grant.
- *sub* – informs the framework about a subject identifier of a user on whom behalf application tries to obtain an access token.
- *enrollment_id* – a place for enrollment identifier of application installation on a device. It should match to one passed in the enrollment process.
- *totp* – a place for TOTP generated using shared secret set-up during the enrollment process.
- *pin_code_encrypted* – user PIN code encrypted using a public key exposed in discovery document of Identity Server.
- *pin_code_encryption_key_id* – thumbprint of a public key used for encryption of a field *pin_code_encrypted*.
- *scope* – informs the framework about a list of Oauth 2.0 scopes pointing resource to which application wants to have access based on access token.

The whole logic is based on the sequence of validations performed by dedicated services:

- *IUserStore* – used to check whether a user pointed in *sub* parameter exists and is active.
- *IEnrollmentService* – used to check whether enrollment pointed in *enrollment_id* parameters exists and is active.
- *IPinCodeService* – used to check whether a user PIN code pointed in *pin_code_encrypted* field is correct.
- *ITotpService* – used to check whether TOTP pointed in *totp* field is correct.

If any of validations fails, standard HTTP 400 status code is returned to the client with no details of the reason. Basic assumption is to not expose details of the reason. Process of client identifier and secret validation is delivered as part of Identity Server 4 framework so grant validator simply "believes" in client data correctness. As the whole framework is IoC driven, the services can be simply overwritten by custom implementation and added to pipeline during the application start-up. Also, thanks to following SOLID rules, responsibility of classes is relatively small so developer can replace atomic pieces of software, if required. Detailed sequence of activities which are performed presents sequence diagram on Figure 5.

The framework logic is driven mainly by configuration which majority is stored in the database. As part of the framework, an integration layer for Microsoft SQL Server [22] has been added to the solution. Also, to prove that the prototype framework meets initial requirements, it has been implemented using two most popular ORMs in author's opinion – Entity Framework Core and NHibernate.

What is also important, the only default option to set the configuration is to execute raw SQL statements against database. To facilitate this process, a set of exemplary migration scripts has been

added to the solution as well. Scripts were written in such a way as to only present concept and minimal required configuration in order to set up a system composed from mobile application, identity server and one resource server.

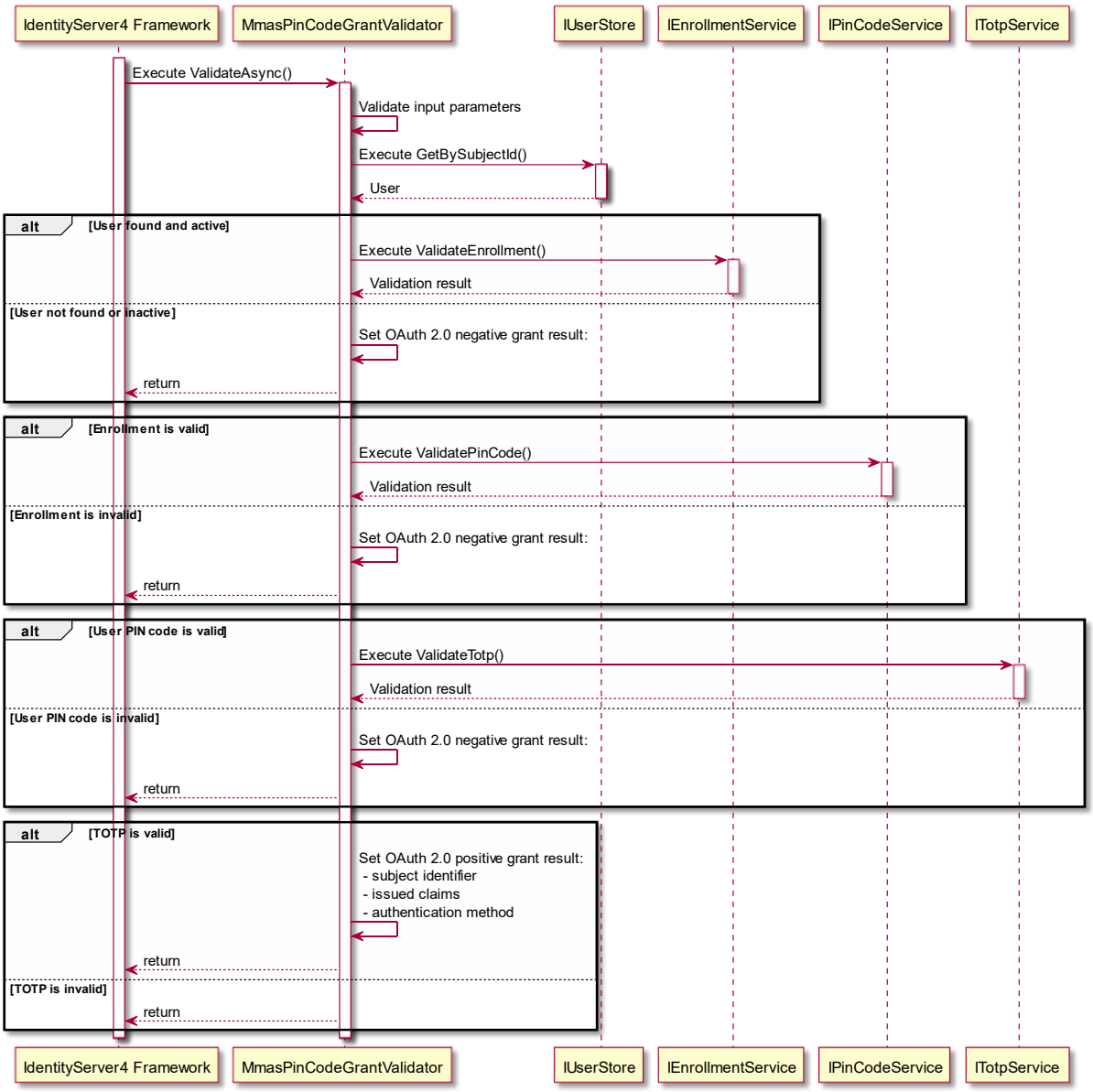


Figure 5: MMAS PIN Code grant logic. Own elaboration.

7.7 Dynamic key material management

Another very important area in MMAS framework is key material management. As it was pointed out in the solution concept and depicted in enrollment and sign-in processes, certificates play a crucial role. They are used during the following steps:

- For signing and validation of issued access tokens.
- For encryption and decryption of TOTP shared secret during enrollment process and the TOTP during sign-in process.
- For encryption and decryption of a user PIN code during enrollment and sign-in processes.

To properly work with key material, it is crucial to properly manage its lifetime. Certificates are valid in specified range of time therefore it is required to perform a rollover of them. If they are not exposed publicly and are used synchronously it is not a problem as the administrator of the service / application can simply provide a new valid certificate when an old one is going to expire. In case of exposure of certificate public keys used either for encryption of values or verification of signatures, it becomes a problem requiring a dedicated approach. It is driven by the fact that system administrator does not control the moment when the client application refreshes the certificate using dedicated JWKS endpoint. Of course, application can force certificate refreshment but there may be a case when the application still has the old public key used for user PIN code encryption when the server side may already have a new one. In such case, sign-in process will fail as the other public key will be used to encrypt value and other one to decrypt it.

The common solution for this and similar problems is a regular process of key material rollover. The high level approach for this issue is presented on Figure 6. It is essential to always keep a set of certificates – let's say 2 – which are in constant use. What is also important, the number of certificates which are exposed to the public clients is always less or equal to overall certificates set count. Key material which should be exposed is named as current one. The remaining key material is not current but it is still valid from technical point of view, so it can be used for either decryption of secrets or verification of JWT tokens signature. Such approach enables turning off for example 1 out of 2 certificates which expiration date is approaching. Once the certificates are marked as not current (action 5 on the diagram), no public client will download them but those clients which downloaded them previously (action 2 on the diagram) may still use them for encryption and validation purposes (action 12,13 on the diagram) – usually 24 hours. After that time administrator may safely remove turned off certificates (action 23 on the diagram) and set up new or temporarily leave remaining ones in place.

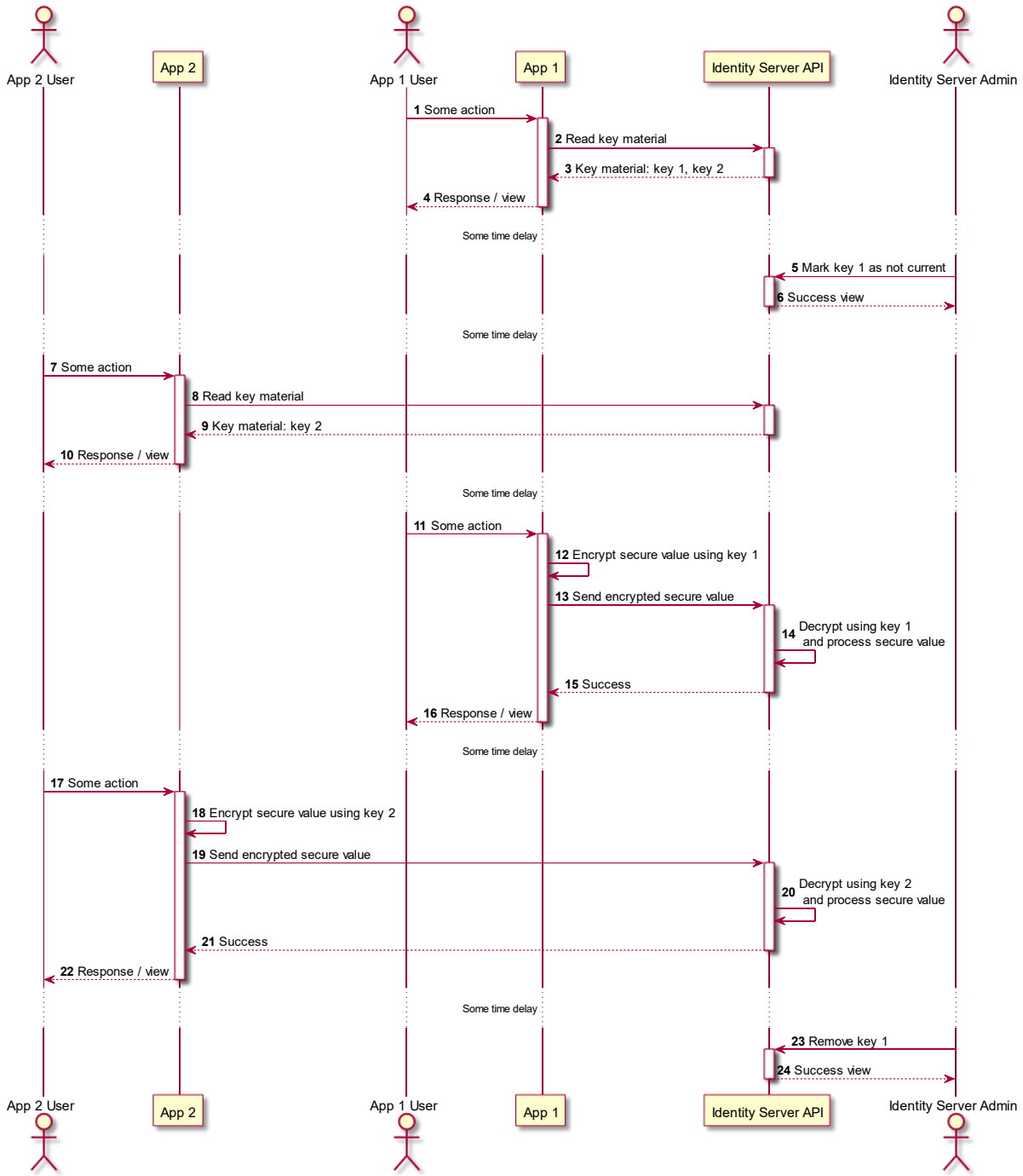


Figure 6: Key material rollover process example. Own elaboration

MMAS framework realisation of this process is driven by interface *ICertificateProvider*. Its role is to provide a simple, plug-in based option for key material retrieval process for the whole server side framework. Provider assumes also existence of configuration model which delivers amount of a time for which retrieved certificates should be cached and a collection of single certificate properties like its location or passphrase which can open it. The certificate location may differ depending on particular use case. It can be a file, machine certificate store or some cloud based certificates storage. Depending on used type of configuration approach, it may require different configuration with variable amount of properties required to obtain it. In order to allow dynamic configuration binding, with no boilerplate code, the base configuration classes have been introduced:

- *BaseCertificateSourceConfiguration* – holds basic configuration for the whole key material. With the aim of adding configuration specific to a particular provider, a new class which inherits from this one needs to be implemented.
- *BaseCertificateSourceConfigurationEntry* – holds basic configuration for a single key. It also requires provider specific implementation.

To prove that the concept is correct and works as expected, two implementations of provider and configuration have been introduced as part of MMAS framework:

- *FileCertificateProvider* – provider which loads key material from files.
- *X509StoreCertificateProvider* – provider which loads key material from machine X509Store.

Listing 19 and Listing 20 present an exemplary configuration for both approaches. As the server side framework targets .NET Core based Web APIs it is delivered in JSON based manner.

Listing 19: *FileCertificateProvider* configuration *appsettings.json*

```
{
  "PinCodeCertificateSourceConfiguration": {
    "CacheExpirationInterval": "00:01:00:00",
    "Entries": [
      {
        "RelativePath": "Certs\\PinCode\\certificate1.pfx",
        "Password": "Aa123456",
        "IsCurrent": "True"
      }
    ]
  },
  "TotpCertificateSourceConfiguration": {
    "CacheExpirationInterval": "00:01:00:00",
    "Entries": [
      {
        "RelativePath": "Certs\\PinCode\\certificate2.pfx",
        "Password": "Aa654321",
        "IsCurrent": "True"
      }
    ]
  }
}
```

Listing 20: *X509StoreCertificateProvider* configuration *appsettings.json*

```
{
  "PinCodeCertificateSourceConfiguration": {
    "CacheExpirationInterval": "00:01:00:00",
    "Entries": [
      {
        "Thumbprint": "C1EC6F328B7B474184C790121D96E745B44FBA4D",
        "IsCurrent": "True"
      }
```

```
    ]
  },
  "TotpCertificateSourceConfiguration": {
    "CacheExpirationInterval": "00:01:00:00",
    "Entries": [
      {
        "Thumbprint": "C1EC6F328B7B474184C790121D96E745B44FBA4D",
        "IsCurrent": "True"
      }
    ]
  }
}
```

The applied approach allows dynamic usage of chosen key material store basing on simple configuration and dependency injection mechanism. Further works on MMAS framework should deliver more provider implementations for the most common scenarios like cloud based stores.

8 MMAS mobile framework

This chapter describes details of mobile part of the MMAS framework. It presents core components and depicts crucial parts of the implementation.

8.1 Datastore

As it was pointed in the design concept, the mobile part of the framework requires secure and effective datastore. In order to achieve it, a dedicated being has been introduced – a database connection factory. This class is responsible for creating and keeping the connection with a datastore which in case of mobile application is an SQLite database, which in fact is a file kept at a device level. This fact implies that approach for using it is different and connection which was once open should be kept in this state unless the logic requires to explicitly close it due to some reason.

In order to achieve it, MMAS framework introduced an interface named *IDatabaseConnectionFactory*. It defines a few methods which are responsible for creation and closure of SQLite connections in a standardised and thread safe manner.

Listing 21: *IDatabaseConnectionFactory* interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Factories
{
    public interface IDatabaseConnectionFactory
    {
        SQLiteConnection GetConnection(string connectionName);

        SQLiteConnection GetConnection(string connectionName, string encryptionKey);

        Task<SQLiteAsyncConnection> GetAsyncConnectionAsync(string connectionName);

        Task<SQLiteAsyncConnection> GetAsyncConnectionAsync(
                                            string connectionName,
                                            string encryptionKey);

        void CloseConnection(string connectionName);

        Task CloseAsyncConnectionAsync(string connectionName);

        void CloseAllConnections();

        Task CloseAllAsyncConnectionsAsync();
    }
}
```

The framework delivers the default implementation of such interface as well. To use it properly, it is required to follow IoC principles and use singleton registration to ensure that only one instance of factory will deliver connection for all parts of the framework.

8.2 Domain

Each software is built on top of some domain. In case of mobile framework, it assumes existence of two beings which are rudiments of the rest of the software – *Parameter* and *User*. Both classes are the models of corresponding tables in semi-secure SQLite database which is created during application start-up:

- *Parameter* (table *Parameters*) – a class which represents a single parameter which can be added, modified and removed during the application runtime. Used for example for storing access and refresh tokens.
- *User* (table *Users*) – a class which represents a single application user. It is created during a process of enrollment and regularly updated during sign-in and sign-out operations to keep the track of user's activity.

Table 5: *User* class properties

Name	**.NET type**	**Description**
Id	Guid	Unique, auto incremental user identifier
SubjectId	string	Unique user identifier with respect to OpenID Connect
EnrollmentId	string	Unique identifier of the application installation at a device
TotpSharedSecret	string	TOTP shared secret used for TOTP codes generation
Username	string	Username
UserDataJson	string	Field for additional properties of a user specific for the application
IsActive	bool	Determines whether a user has been activated
IsEnrolled	bool	Determines whether a user performed enrollment
IsLoggedIn	bool	Determines whether a user is currently logged in
CreatedAtUtc	DateTime	Datetime at UTC time zone when parameter has been initially created
LastUpdatedAtUtc	DateTime	Datetime at UTC time zone when parameter has been recently updated

Table 6: *Parameter* class properties

Name	**.NET type**	**Description**
Id	Guid	Unique, auto incremental parameter identifier
UserId	Guid	Unique identifier of a user to whom it belongs
Key	string	Parameter key
Value	string	Parameter value
CreatedAtUtc	DateTime	Datetime at UTC time zone when parameter has been initially created
LastUpdatedAtUtc	DateTime	Datetime at UTC time zone when parameter has been recently updated

Following the design concept, the dedicated repositories have also been implemented to separate database related operations from the business logic. The corresponding interfaces are presented on Listing 22 and Listing 23.

Listing 22: *IParameterRepository* interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Repositories
{
    public interface IParameterRepository
```

```
    {
        void SaveOrUpdate(Guid userId, string key, string value, string
serviceAccountUsername = MmasConstants.Mobile.Constants.ServiceAccountUsername);

        Task SaveOrUpdateAsync(Guid userId, string key, string value,
string serviceAccountUsername = MmasConstants.Mobile.Constants.ServiceAccountUsername);

        Task<string> GetByKeyAsync(Guid userId, string key);

        string GetByKey(Guid userId, string key);
    }
}
```

Listing 23: *IUserRepository* definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Repositories
{
       public interface IUserRepository
       {
              IList<User> ListActiveUsers();

              Task<IList<User>> ListActiveUsersAsync();

              void SaveOrUpdate(User user);

              Task SaveOrUpdateAsync(User user);

              void Delete(User user);

              Task DeleteAsync(User user);

              Task<User> GetBySubjectIdAsync(string subjectId);

              User GetBySubjectId(string subjectId);

              Task<User> GetLoggedInAsync();

              User GetLoggedIn();

              Task<User> GetByUsernameAsync(string username);

              User GetByUsername(string username);
       }
}
```

With the aim of proper management of facets which MMAS framework addresses, the author designed several beings with their default implementations:

- *IHttpService* – defines service responsible for communication with server API. It adds additional layer over default .NET *HttpClient* responsible for areas like access token addition to request.
- *IHttpServiceWithCheck* – defines service with same responsibility as *IHttpService* but also introduces an additional layer responsible for checking identity and discovery process.
- *IHttpAuthService* – defines a service responsible for access token retrieval for HTTP requests. It is also the only interface which requires application specific implementation.
- *IPreferenceService* – defines service responsible for access to device preferences.

- *ISecureStorageService* – defines service responsible for access to device specific secure storage. Secure means here a store which is available only when a device is unlocked by a user.
- *IDeviceInfoService* – defines service responsible for access to information about a device like platform, OS version and a few more.
- *IEnrollmentService* – defines service responsible for enrollment of the application installation. It generates enrollment identifier, performs communication with the server and persists response in dedicated place.
- *ITotpService* – defines service responsible for computation of TOTP codes based on provided shared secret.
- *IDiscoveryService* – defines service responsible for performing identity server discovery.
- *IIdentityService* – defines service responsible for Authn and Authz areas like discovery process, sign-in and sign-out functionalities.
- *IIdentityValidationService* – defines service responsible for validation of user identity as well as its refresh, if expired.
- *ITokenDataStore* – defines service responsible for persistence of token data.
- *IMetadataService* – defines service responsible for obtaining metadata about the application. Details about its intended use will be described in subsequent chapters.
- *ISecretService* – defines service responsible for creation and management of client and database secrets.

8.3 Application bootstrapping

With a view to properly managing application start-up, MMAS framework has been enhanced with additional being which is executed each time when the application is run. This being follows concept of bootstrapping and performs a few important operations:

- Registers configuration in IoC container as the singleton.
- Creates semi-secure database with tables if they do not exist.
- Sets up the certificate pinning logic.
- Performs discovery and persists response properties in relevant places.
- If a user was logged during last application run, it determines whether their identity requires refresh and prompts a user to enter PIN code if required.

Interface defining bootstrapper is presented on Listing 24.

Listing 24: *IAppBootstrapper* interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Bootstrappers
{
    public interface IAppBootstrapper
    {
        Task Configure(AppBootstrappRequest appBootstrappRequest);

        Task<AppBootstrappResult> Initialize();

        Task<AppBootstrappResult> ConfigureAndInitialize(
              AppBootstrappRequest appBootstrappRequest);
    }
}
```

Such definition gives an option to fire bootstrapper in two manners depending on the particular use case. One option is to use method *ConfigureAndInitialize* which performs all required operations. This is a default variant. On the other hand, there is a second option which requires subsequent calls to methods *Configure* and *Initialize*. Such approach gives a possibility to perform additional operations between the configuration and initialisation like additional required registration in IoC container.

Majority of operations which are performed during bootstrap of the application require configuration specific for the application. Additionally, the Xamarin Forms does not have built in IoC mechanism and developers may use many different frameworks used for dependency injection. To handle this area in a proper way, the configuration has been extended with a few delegates which are used by MMAS framework to interact with IoC container without knowledge which one has been used by a developer.

In order to ensure that Authn and Authz areas are properly handled in the mobile application, it is also required to manage correctly user identity and its expiration. As the MMAS framework must deal with this facet in a few places, proper reaction is also necessary, for instance, when the access token expires and only option is to prompt a user to sign in. Such an event should be application specific and to enable such option MMAS framework configuration requires additional delegates which will be fired each time when sign-in is needed.

The undermentioned list presents all configurable properties of the MMAS framework which can be passed via *AppBootstrappRequest* to MMAS framework configuration and initialisation logic:

- *IssuerEndpoint* (string) – represents the base endpoint of the issuer, what means the identity server API.
- *EnrollmentEndpoint* (string) – represents the base endpoint of the API being responsible for enrollment process.
- *AppClientId* (string) – unique identifier of the application in Identity Server datastore which is used to determine associated configuration. There is no option to set here corresponding *AppClientSecret* as in case of mobile applications which are public clients there is no option to keep such a value safely.
- *IssuerDiscoveryEndpoint* (string) – represents endpoint used for process of Identity Server discovery. If not explicitly set, it is evaluated to default value using IssuerEndpoint.
- *DiscoveryResultExpiration* (TimeSpan) – represents amount of time after which the result of discovery process is kept at a device level. If not explicitly set, the default value of 1 hour is set.
- *TryRefreshIdentityWhenUnauthorized* (bool) – determines whether MMAS framework should try to refresh identity when unauthorised response is reported from Resource Server API or when access token simply expires.
- *IsPinCodeGrantEnabled* (bool) – determines whether PIN code grant is enabled. Thanks to it one may disable PIN code grant hence the enrollment process as well.
- *LocalPinCodeExpiration* (TimeSpan) – represents amount of time after which a PIN code value is kept in a device secure storage in a hashed form. Local copy is used when a user is prompted to enter a PIN code and access token is still valid.
- *PinnedPublicKeys* (string list) – represents a list of pinned server APIs public keys used in certificate pinning process.
- *UnauthorizedCallbackForPinCodeFlow* (delegate) – callback used when sign-in with user's PIN code ends with unauthorised result.
- *UnauthorizedCallbackForPasswordFlow* (delegate) – callback used when sign-in with user's password ends with unauthorised result.

- *IocRegisterSingletonCallbackLazy* (delegate) – callback used for interaction with the IoC container to register lazy singleton. Lazy means here that singleton is created once it is resolved first time from the container.
- *IocRegisterSingletonCallbackInstance* (delegate) – callback used for interaction with the IoC container to register standard singleton. It means the instance of a class is created immediately.
- *IocRegisterTransientCallback* (delegate) – callback used for interaction with the IoC container to register dependency in a transient manner. It means that each resolution gives a new instance of the registered class.
- *IocResolveCallback* (delegate) – callback used for interaction with the IoC container to resolve registration to obtain an instance of the class in manner depending on the registration approach.

The Listing 25 presents bootstrapper exemplary usage in Xamarin Forms based application with MvvmCross [23] framework used for MVVM pattern support. It also presents the methodology of setting up configuration parameters mentioned above.

Listing 25: *IAppBootstrapper* exemplary usage with MvvmCross framework

```
// … logic …
IMvxNavigationService navigationService = Mvx.IoCProvider.Resolve<IMvxNavigationService>();

AppBootstrappRequest appBootstrappRequest = new AppBootstrappRequest()
{
    Environment = "dev",
    IssuerBaseEndpoint = "https://mmas-auth.azurewebsites.net",
    AppClientId = "3c57ca1683f54c90bf99f69e00fe3aa1",
    EnrollmentBaseEndpoint = "https://mmas-task-kit.azurewebsites.net",
    IssuerDiscoveryEndpoint = null,
    DiscoveryResultExpiration = new TimeSpan(0, 1, 0, 0),
    TryRefreshIdentityWhenUnauthorized = true,
    IsPinCodeGrantEnabled = true,
    LocalPinCodeExpiration = new TimeSpan(0, 0, 5, 0),
    PinnedPublicKeys = null, // Set here pinned public keys
    UnauthorizedCallbackForPinCodeFlow =
       async () => await navigationService.Navigate<IdentityPinCodeLoginViewModel>(),
    UnauthorizedCallbackForPasswordFlow =
       async () => await navigationService.Navigate<IdentityPasswordLoginViewModel>(),
    IocRegisterSingletonCallbackLazy =
       (x, y) => { Mvx.IoCProvider.LazyConstructAndRegisterSingleton(x, () =>
       Mvx.IoCProvider.IoCConstruct(y)); },
    IocRegisterSingletonCallbackInstance =
       (x, y) => { Mvx.IoCProvider.RegisterSingleton(x, y); },
    IocRegisterTransientCallback = (x, y) => { Mvx.IoCProvider.RegisterType(x, y); },
    IocResolveCallback = (x) => Mvx.IoCProvider.Resolve(x)
};

await this._coreAppBootstrapper.Configure(appBootstrappRequest);
```

```
appBootstrappRequest.IocRegisterSingletonCallbackLazy(typeof(IHttpAuthService),
typeof(HttpAuthService));

AppBootstrappResult appBootstrappResult = await this._coreAppBootstrapper.Initialize();

isAuthenticated = appBootstrappResult.IsAuthenticated;
// … logic …
```

8.4 Enrollment, sign-in and sign-out

The core and the most important part of the MMAS framework mobile part are methods responsible for enrollment, sign-in and sign-out related processes. To standardise the methodology of performing such kind of operations, the *IIdentityService* and *IEnrollmentService* interface has been introduced. The definitions of these interfaces are presented on Listing 26 and Listing 27.

Listing 26: *IIdentityService* interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Services
{
    public interface IIdentityService
    {
        Task<T> GetUserDataAsync<T>()
              where T : BaseUserDataDto;

        Task<User> SignInUsingPasswordAsync<T>(SignInUsingPasswordRequest request)
              where T : BaseUserDataDto;

        Task<User> SignInUsingPinCodeAsync<T>(SignInUsingPinCodeRequest request,
              bool isPostEnrollmentLogin) where T : BaseUserDataDto;

        Task SignOutAsync();
    }
}
```

Listing 27: *IEnrollmentService* interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Services
{
    public interface IEnrollmentService
    {
        Task PerformEnrollmentAsync(EnrollRequest request);
    }
}
```

Methods exposed by these interfaces constitute the core API exposed by MMAS mobile framework. Each of them plays crucial role and incorporates everything what must be done to perform the logic in a secure manner:

- *SignInUsingPasswordAsync* – allows performing sign-in operation using standard flow with a username and a password.

- *PerformEnrollmentAsync* – allows performing enrollment operation.
- *SignInUsingPinCodeAsync* – allows performing sign-in operation using new MMAS user PIN code flow. Method can be executed in two manners – just after the enrollment as the continuation of it and in the standard sign-in scenario.
- *GetUserDataAsync* – allows retrieving user data. Method is generic what means that it can retrieve standard information like a subject identifier as well as a custom one.
- *SignOutAsync* – allows performing sign-out operation at a device level.

Figure 7, Figure 8 and Figure 9 present *SignInUsingPasswordAsync*, *SignInUsingPinCodeAsync* and *SignOutAsync* logic. All diagrams have been simplified to present the crucial logic.

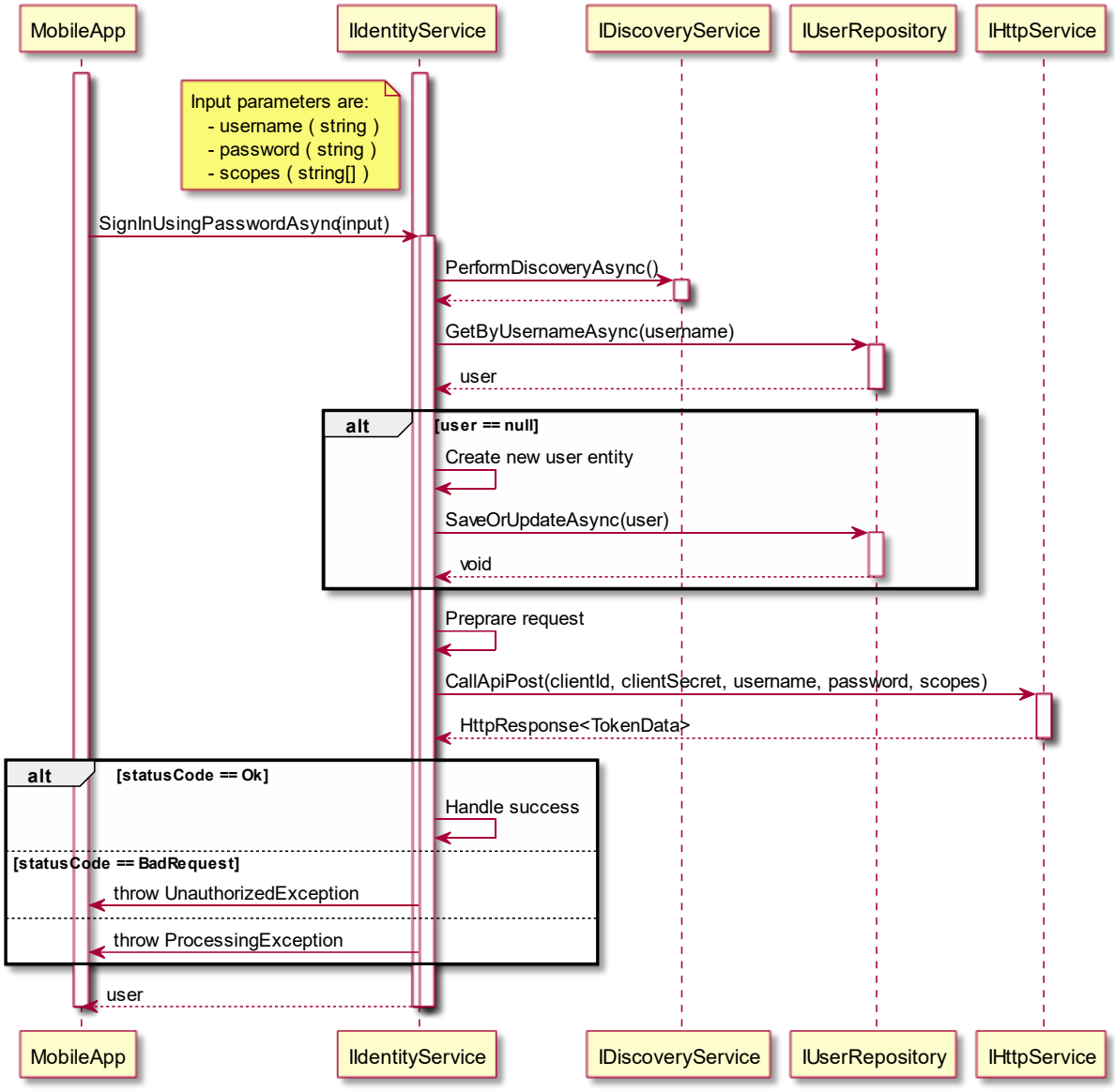


Figure 7: *SignInUsingPasswordAsync* method logic. Own elaboration

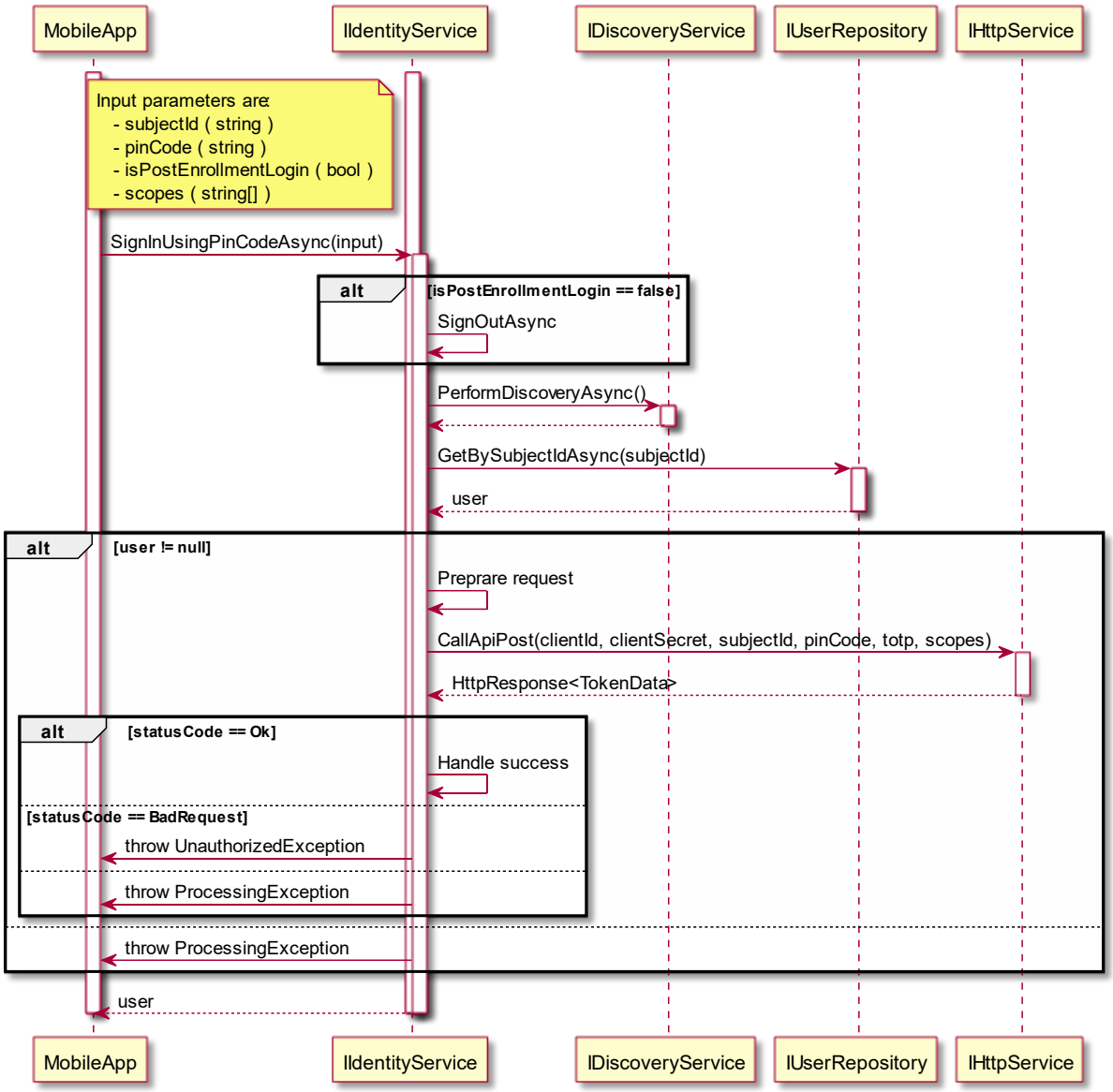


Figure 8: *SignInUsingPinCodeAsync* method logic. Own elaboration

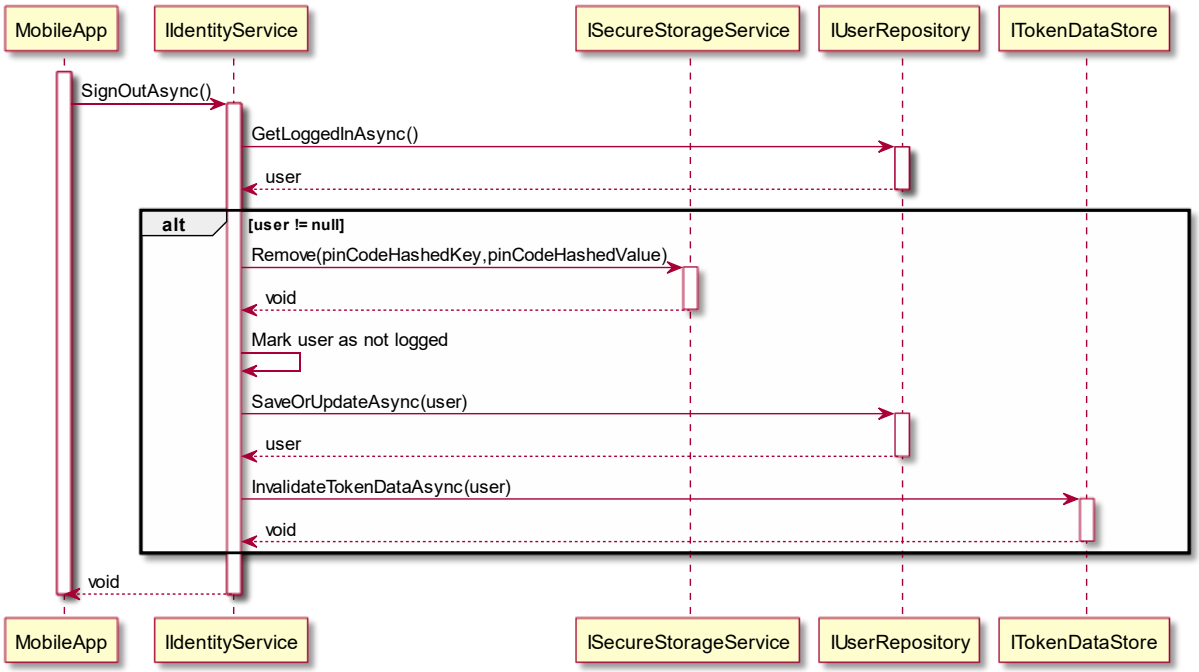


Figure 9: *SignOutAsync* method logic. Own elaboration

8.5 Secrets obfuscation

As it is said in OpenID Connect documentation [2] and what is recommended by [5] Aaron Parecki in his book, the mobile client application should not use the client secret for communication with Identity Server API. It is totally correct and valid recommendation as to achieve it the secret needs to be kept in the code in a plain text form what is a non-secure approach.

In author's opinion there is an option to improve this area by applying special approach which requires some captious logic. There are two general assumptions which may cause that client secret may be used:

- Secret should be composed of two values – base, obfuscated secret and a salt created from application runtime environment.
- Base secret should be delivered separately for each environment in form of native C++ library attached to the solution. The library should be recompiled with different inputs for each environment. Such library once intercepted from the test environment for instance cannot be used to obtain a secret from the production environment.
- The library delivering base secret should be obfuscated to the maximum to hamper its reading and interpretation to the utmost.
- The client secret cannot be used to perform any operation standalone, what in OpenID Connect language means that the client credentials grant cannot be used standalone.

With such assumptions, usage of client secret could be used as an additional security layer for communication with backend API. Of course, no matter how well it will be obfuscated there is always a risk that potential attacker may intercept the library and crack the code. Nevertheless, intercepted secret is only one of many security layers therefore losing it will weaken communication safety but not break it.

The exemplary approach for such kind of obfuscation may be based on usage of algorithms presented by Donald E. Knuth in his book '*The art of computer programming*' [26]. In section 3.6 he presents methods of random values generation using efficient C/C++ language based algorithms. In general, the logic should use random methods to implicitly generate a static value. Additional factor to hide real logic may be also building a code which returns an expected secret only when the algorithm is executed number of times. Another meaningful factor may be also a misleading naming of variables.

As the MMAS framework is only a Proof of Concept (POC), this area has been replaced with temporary and fake C# implementation. Further works should consider building logic presented above in a way which will deliver complete solution that can be attached as a library to any project or at least a guide how to perform it properly. Listing 28 presents conceptual *IMetadataService* interface which defines a few methods responsible for delivery of metadata related to the application. The most important method is *GetThumbprint()* which in fact should deliver the abovementioned secret in an obfuscated way.

Listing 28: IMetadataService interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Services
{
	public interface IMetadataService
	{
		string GetBuild();

		string GetBranch();

		string GetLicense();

		string GetKey();

		string GetHash();

		string GetThumbprint();
	}
}
```

For the sake of not forcing potential users to use this logic, client secret maturity can be simple turned off by appropriate configuration at the server side which is delivered out of the box by Identity Server 4 framework.

8.6 Navigation handler

With the focus on ensuring that mobile application is secured in all use cases there is a need to intercept the navigation between the application views. It is required as between subsequent navigations there is a passage of time which leads to expiration of access token obtained during sign-in or refresh grant processes. There is a need to handle each navigation event to check whether a user should be prompted to sign in again or whether the application should perform background refresh of the access token (if it is enabled by the configuration). In case of Xamarin Forms applications with MVVM design pattern there are many approaches to intercept the event of navigation between screens / pages. As the approach for navigation may be realised in various forms depending on MVVM implementation, it is hard to deliver software which manages it out of the box. In return, the author has decided to deliver a being which depending on the fact whether the user PIN code flow is enabled or not executes relevant delegates passed in configuration, what means either *UnauthorizedCallbackForPinCodeFlow* or *UnauthorizedCallbackForPasswordFlow* property.

Without regard for MVVM approach used implementation, the management of navigation events is required by MMAS framework and this area should be taken into account to use it properly. The Listing 29 presents *INavigationHandler* interface definition and on Listing 8.10 one can see its usage in the exemplary application built as part of the prototype.

Listing 29: INavigationHandler interface definition

```
namespace Pjatk.s16085.Mmas.Mobile.Domain.Interfaces.Handlers
{
    public interface INavigationHandler
    {
        Task HandleNavigationEventAsync(bool isTargetProtected);
    }
}
```

As it can be seen, the method responsible for handling a navigation event accepts parameter determining whether the target of navigation is protected. In other words, this parameter points MMAS framework if the access token validity should be checked before navigation in order to refresh it or prompt a user to re-enter credentials.

Listing 30: *INavigationHandler* usage in MvvmCross based prototype

```
#region Lifecycle methods

public override async Task Initialize()
{
    await base.Initialize();

    if (this.NavigationHandler != null)
    {
        await this.NavigationHandler.HandleNavigationEventAsync(this.IsProtected);
    }

    await Task.CompletedTask;
}

public override void ViewAppearing()
{
    base.ViewAppearing();

    if (this.NavigationHandler != null)
    {
        this.NavigationHandler.HandleNavigationEventAsync(this.IsProtected);
    }
}

#endregion Lifecycle methods
```

As covered on Listing 30, the management of navigation events is fired in two cases:

- By *Initialize* method - when a particular view and view model being a target of the navigation is initialised.
- By *ViewAppearing* method - when a particular view is about to appear as part of the navigation or after application resuming.

One should get familiar with MvvmCrosss framework documentation [23] to understand in full meaning of these methods.

9 MMAS – exemplary system

This chapter presents usage of the MMAS framework and exemplary system built on top of it.

9.1 Task Kit – solution structure

TaskKit is an exemplary system built using MMAS frameworks. One can distinguish the following projects with their responsibilities:

- **Example.Common** – contains classes which usage is common for all other projects.
- **Example.Data** – contains datastore integration layer.
- **Example.WebApi.Auth** – contains Web API representing Identity Server.
- **Example.WebApi.Core** – contains Web API representing Resource Server.
- **Example.Mobile.App** – contains mobile application common layer.
- **Example.Mobile.App.Droid** – contains mobile application Android specific layer.
- **Example.Mobile.App.iOS** – contains mobile application iOS specific layer.

As the system has been created only for presentation purposes, the naming and relations have been simplified; for example, project **Example.Data** contains logic for both Entity Framework Core and NHibernate integration. Figure 10 presents all code dependencies.

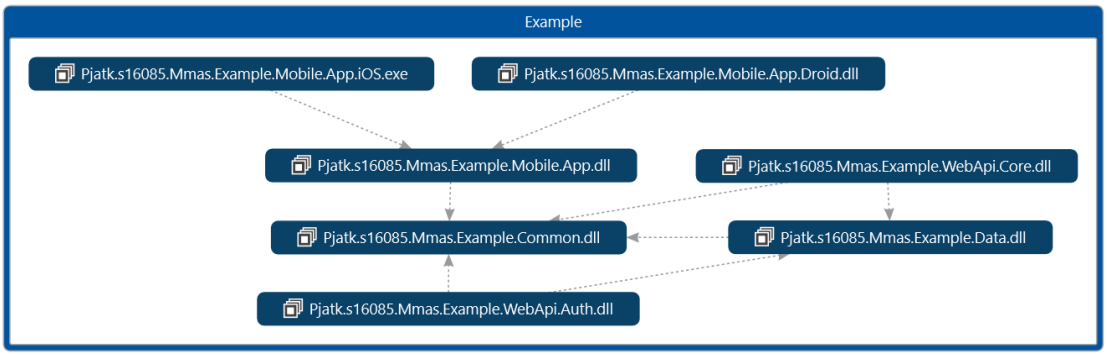


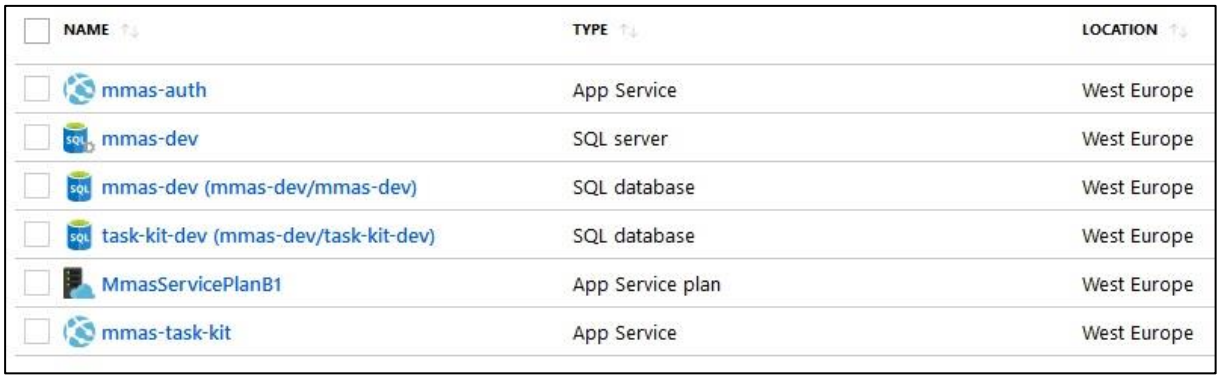
Figure 10: Task Kit system structure. Generated using [17]

9.2 Task Kit – Azure App Services

The server side infrastructure is hosted in Microsoft Azure cloud. The whole consists of three logical components:

- **mmas-auth** – an application which plays a role of the Identity Server which is the ASP.NET Core Web API hosted using Azure App service.
- **mmas-task-kit** – an application which plays a role of the Identity Server which is also the ASP.NET Core Web API hosted using Azure App service.
- **mmas-dev** – the SQL Server instance with two databases:
    - **mmas-dev** – configuration and operational database of the Identity Server API.
    - **task-kit-dev** – operational database of the Resource Server API.

Figure 11 presents the list of all Azure resources composing server side infrastructure of the exemplary system.



NAME	TYPE	LOCATION
mmas-auth	App Service	West Europe
mmas-dev	SQL server	West Europe
mmas-dev (mmas-dev/mmas-dev)	SQL database	West Europe
task-kit-dev (mmas-dev/task-kit-dev)	SQL database	West Europe
MmasServicePlanB1	App Service plan	West Europe
mmas-task-kit	App Service	West Europe

Figure 11: Task Kit cloud infrastructure. Own elaboration

9.3 Task Kit – Identity Server API configuration

Proper working of the application together with the Identity Server and Resource Server API require proper configuration of the standard entities defined by OpenID Connect and OAuth 2.0 protocols [2][3]. So that it is achieved, a few database entries have been added to Identity Server database tables. The list below describes the most important tables being a part of the configuration:

- The *Client* – entry in this table represents single client application which can process Oauth 2.0 grants within Identity Server API in order to obtain access tokens.
- The *ClientSecret* – entry in this table represents single *Client* secret. Together with the *ClientId* it represents client application credentials in the light of OpenID Connect protocol.
- The *ClientGrantType* – entry in this table represents allowed OAuth 2.0 grant type which can be used by the *Client*.
- The *ApiResource* – entry in this table represents single API resource exposed via Resource Server. In view of OpenID Connect it can be identified as the *audience*.
- The *ApiScope* – entry in this table represent single API scope.

Aforementioned entities and their meaning are strongly connected with details of the OpenID Connect and OAuth 2.0 protocols realised by the Identity Server 4 framework. One should become familiar with the documentation [2][3][8] to get a good understanding of these entities meaning.

9.4 Task Kit – Xamarin Forms Application

The mobile application which is a presentation layer of the exemplary system is a simple application targeting area of task management in form of a simple TODO list. The system assumes a few basic use cases which can be done by a user:

- Sign-in using password.
- Perform enrollment.
- Sign-in using PIN code.
- Display list of all TODO items.
- Display details of a single TODO item.
- Create a new TODO item.
- Modify an existing TODO item.
- Mark an existing TODO item as done.
- Delete an existing TODO item.

The system does not support the process of a user registration as the whole framework was designed to support sign-in scenarios. Due to this fact, the author assumed that a user must have an existing account to sign in Task Kit application. As the framework delivers only the API, not the complete functionality, it is also required to orchestrate subsequent calls in the code of TaskKit application. Of course, there are no rigid rules which determine how to use API but some schema has to be met to achieve the goal. The activity diagram presented on Figure 12 presents at a high level activity performed after application start-up. Subsequently, figures from 9.4 to 9.13 show scenario from the moment of application initialisation until positive sign-in and presentation of TODO items list. Each view is presented for both the Android (on the left side) and the iOS (right side) version of the application.

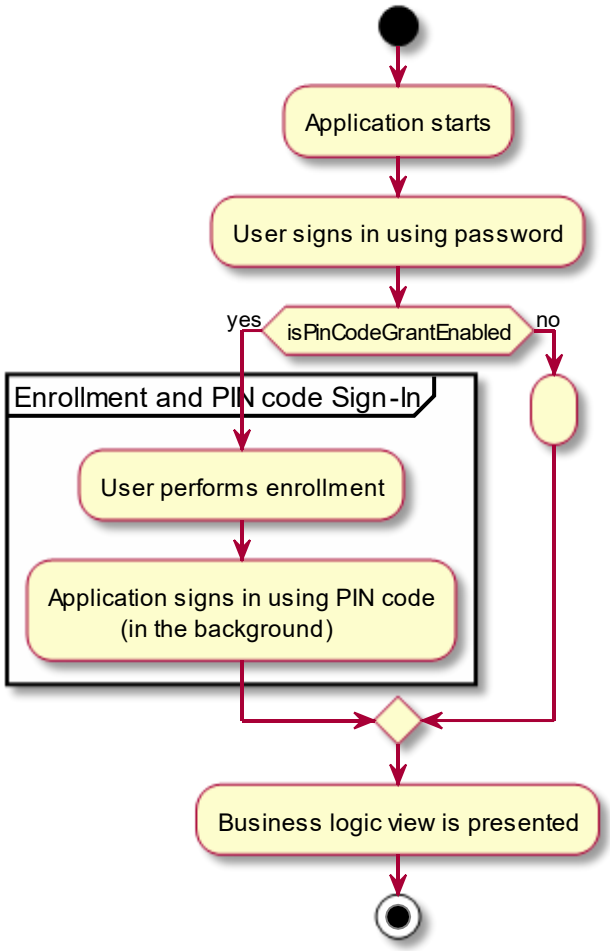


Figure 12: Application start-up. Own elaboration

The *Loading View* presented on Figure 13 is the view shown to a user during application initialisation. It is intended to indicate that application is performing some long-running processes. In fact, the application simply executes MMAS *IAppBootstrapper* class which looks after all required activities that must be done.

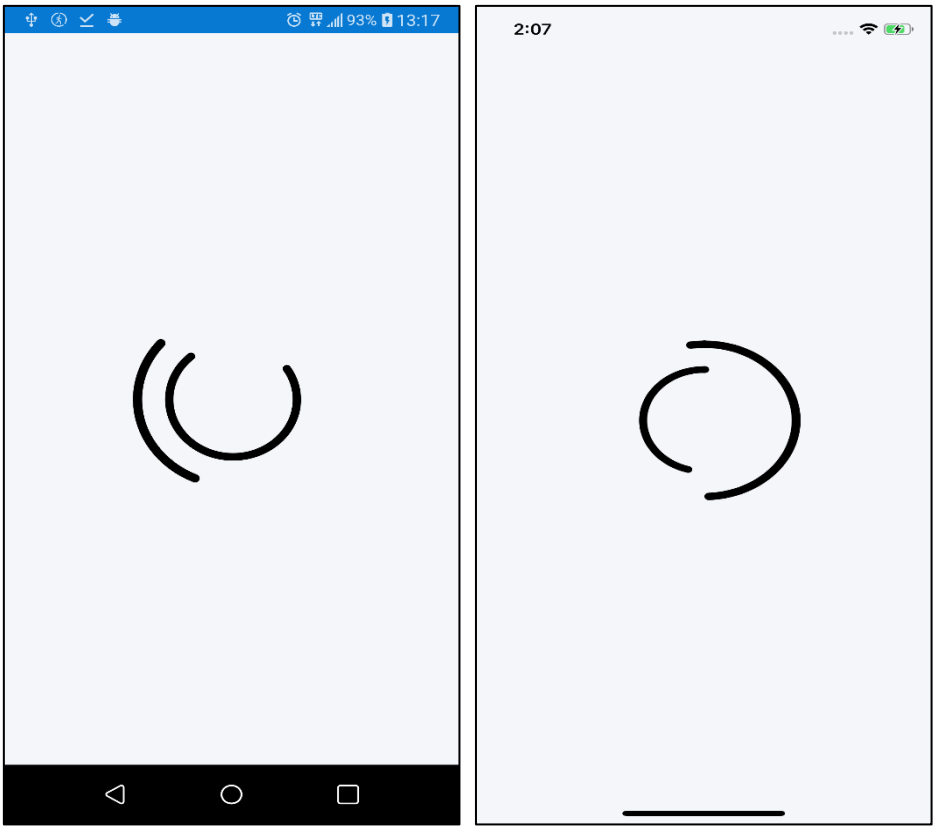


Figure 13: *Loading View*. Own elaboration

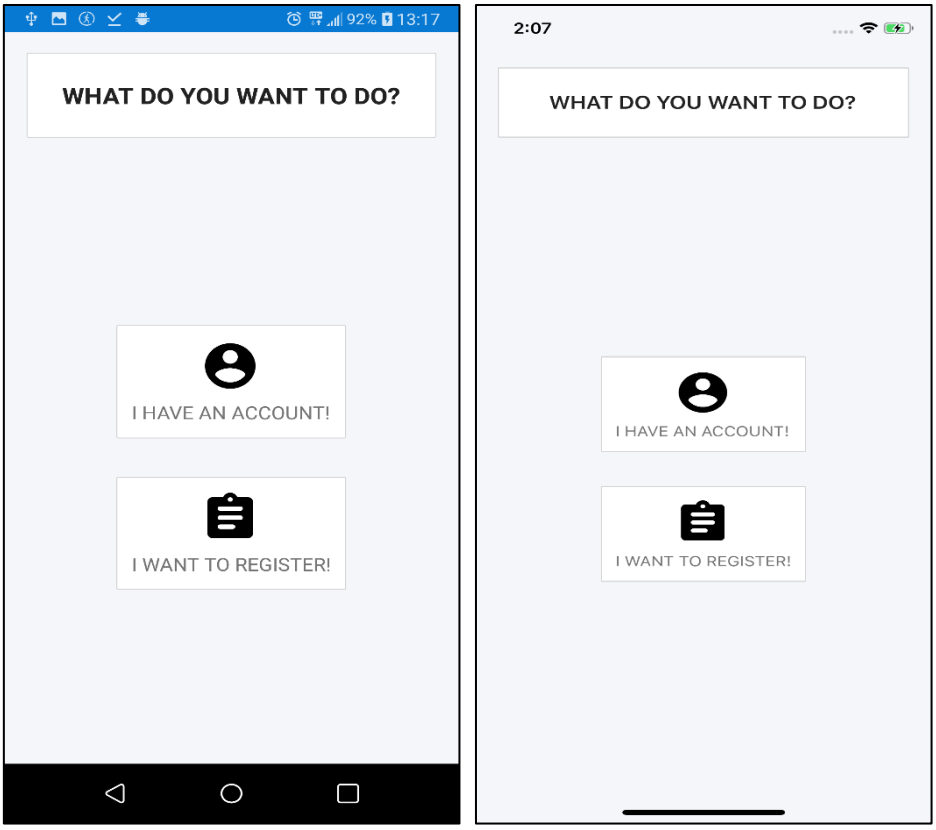


Figure 14: *Identity Initial View*. Own elaboration

The second view which is presented to a user is the *Identity Initial View (*Figure 14). This view simply presents possible paths which may be taken:

- Path for an existing user with the existing account in the service.
- Path for a new user. This area is in fact a dead-end part of the application as the author's motivation was to present MMAS framework usage only for sign-in scenarios.

Figure 15 presents the view at which a user can sign in the application using a username and a password. The Figure 16 presents the same view when a user has submitted their credentials. In the background there are performed appropriate activities which are carried out by the MMAS framework. As the result of the operation, application receives an access token with a permission to perform enrollment. Listing 31 presents the code which is being executed under the hood.

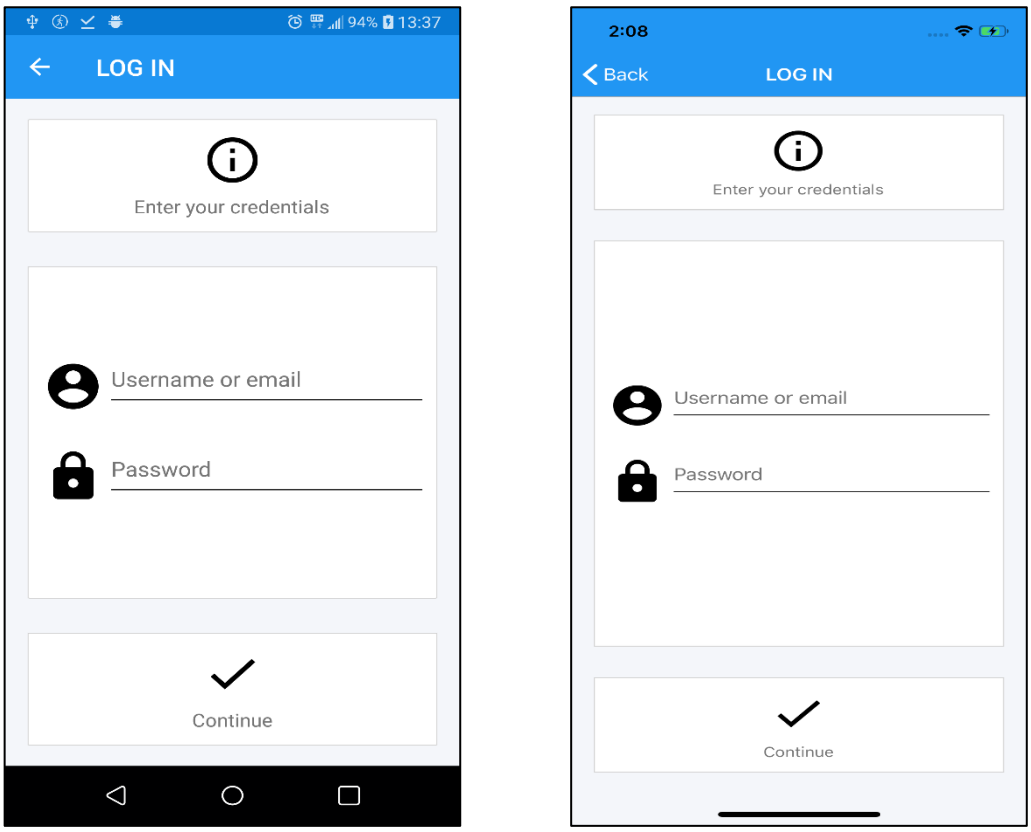


Figure 15: *Identity Password Login View* – empty. Own elaboration

Listing 31: Logic executed under the button *Continue* from Figure 16

```
private async Task ContinueAsync()
{
    try
    {
        base.MakeBusy();
        this.Username.Validate();
        this.Password.Validate();

        if (this.Username.IsValid && this.Password.IsValid)
        {
```

```
            SignInUsingPasswordRequest request = new SignInUsingPasswordRequest()
            {
                Username = this.Username.Value,
                Password = this.Password.Value,
                Scopes = new List<string>()
                {
                    OidcConstants.StandardScopes.Profile,
                    OidcConstants.StandardScopes.OpenId,
                    OidcConstants.StandardScopes.OfflineAccess
                }
            };

            User user
= await this._identityService.SignInUsingPasswordAsync<UserDataDto>(request);

// Further logic and navigation to next view

        }
    }
    catch (UnauthorizedException)
    {
        this.ValidationSummary
              = AppStrings.IdentityPasswordLoginView_ValidationErrorMessageInvalidText;
        this.IsValidationSummaryVisible = true;
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
        await PopupNavigation.Instance.PushAsync(new ErrorPopup());
    }
    finally
    {
        base.MakeNotBusy();
    }
}
```

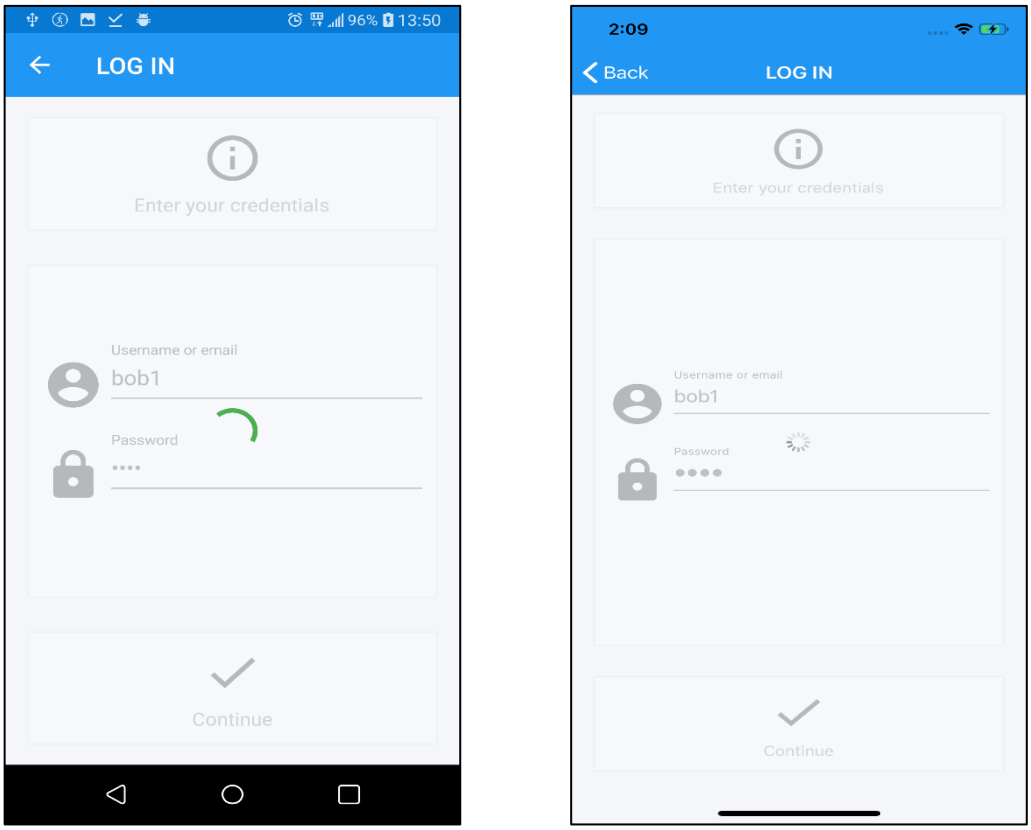


Figure 16: *Identity Password Login View* – processing. Own elaboration

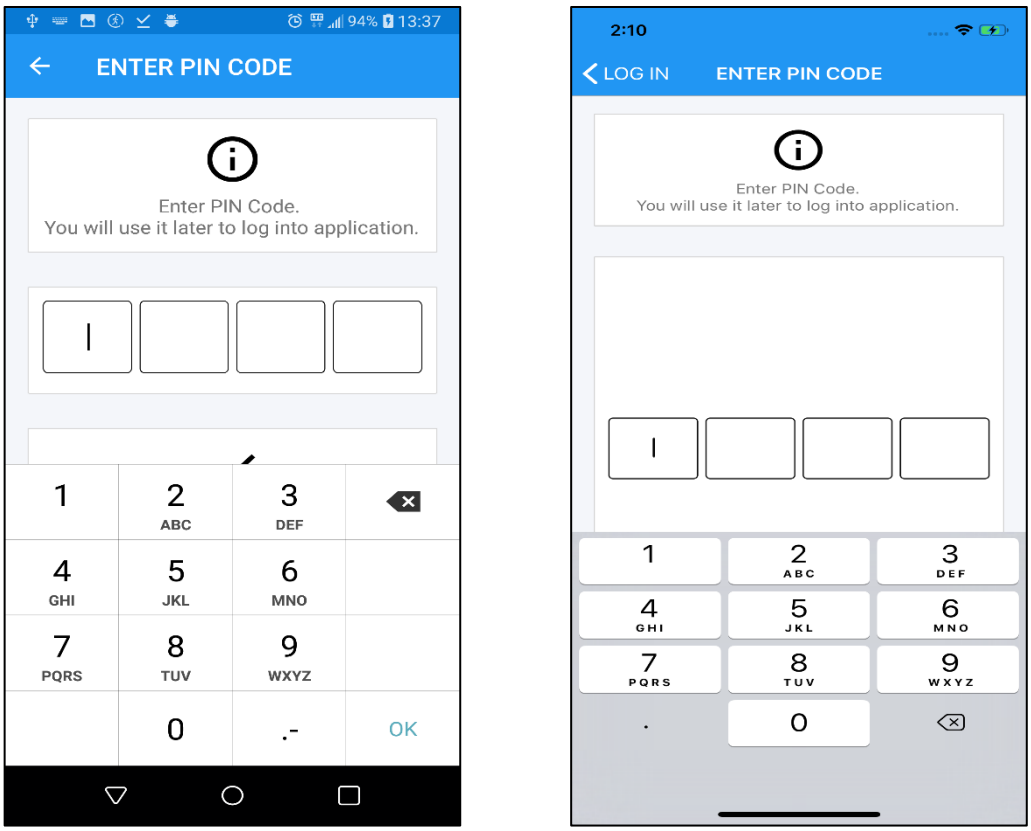


Figure 17: *Enrollment Pin Code View* – part 1. Own elaboration

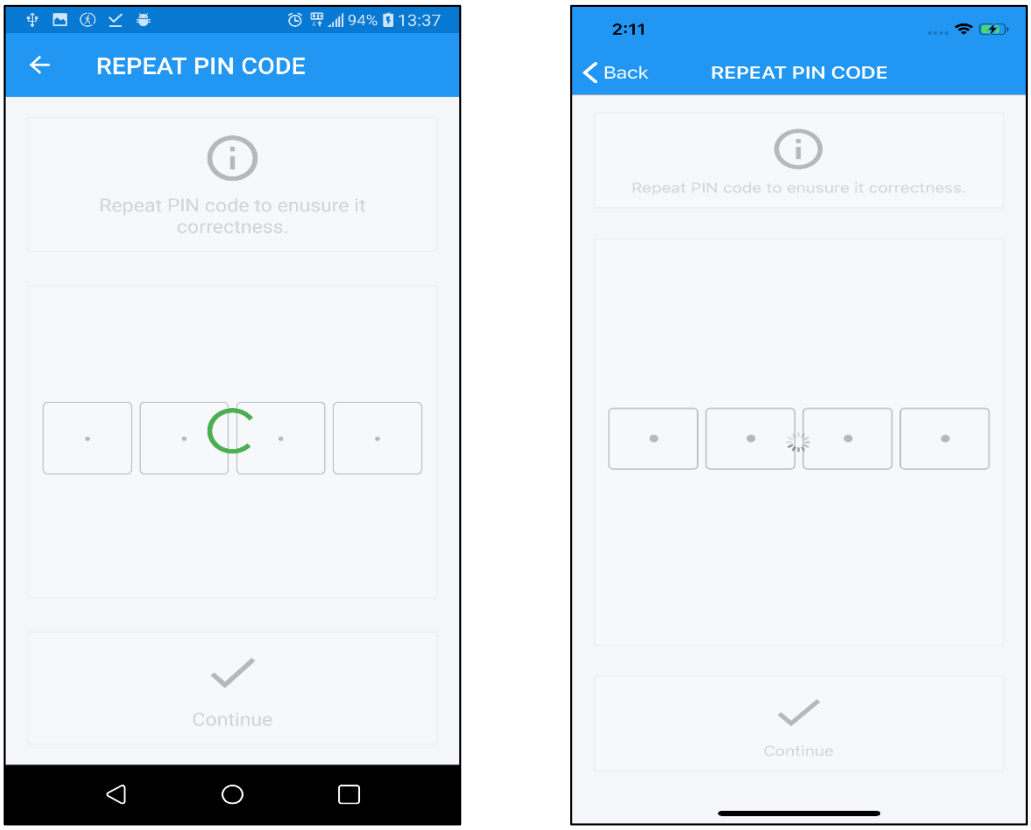


Figure 18: *Enrollment Pin Code View* – part 2. Own elaboration

After positive sign-in, a user can perform enrollment (Figure 17 and Figure 18). From user's perspective it is standard set-up of a PIN code, but from a developer's perspective it is simple subsequent execution of the methods *IEnrollmentService.PerformEnrollmentAsync(enrollRequest)* and *IIdentityService. SignInUsingPinCodeAsync(signInRequest, true)* – Figure 18. Under the hood, MMAS framework sets up required data, perform encryption using key material downloaded during application initialisation and finally posts data to the server. As a result, application obtains a new access token which allows performing operations on protected resources on behalf of a user. The token which is returned has access to the following OAuth 2.0 API scopes:

- *api.taskkit.todoitems.read.own* – api scope which allows reading of user's TODO items.
- *api.taskkit.todoitems.create.own* – api scope which allows creating new user's TODO items.
- *api.taskkit.todoitems.patch.own* – api scope which allows modifying existing user's TODO items.
- *api.taskkit.todoitems.delete.own* – api scope which allows deleting existing user's TODO items.

Figure 19, Figure 20 and Figure 21 present selected views of the application after sign-in and enrollment processes are completed. They show basic scenarios with read (scope *api.taskkit.todoitems.read.own*) and create (scope *api.taskkit.todoitems.create.own*) scenarios.

Listing 32: Logic executed under the button *Continue* from Figure 18

```
private async Task ContinueAsync()
{
    try
    {
```

```
            base.MakeBusy();

            this.PinCode.Validate();

            if (this.PinCode.IsValid && this.PinCode.Value.Equals(this._parameters.PinCode))
            {
                EnrollRequest enrollRequest = new EnrollRequest()
                {
                    PinCode = this._parameters.PinCode
                };

                await this._enrollmentService.PerformEnrollmentAsync(enrollRequest);

                SignInUsingPinCodeRequest signInUsingPinCodeRequest
                  = new SignInUsingPinCodeRequest()
                {
                    SubjectId = this._parameters.SubjectId,
                    PinCode = this.PinCode.Value,
                    Scopes = new List<string>()
                    {
                        OidcConstants.StandardScopes.Profile,
                        OidcConstants.StandardScopes.OpenId,
                        OidcConstants.StandardScopes.OfflineAccess,
                        "api.taskkit.todoitems.read.own",
                        "api.taskkit.todoitems.create.own",
                        "api.taskkit.todoitems.write.own",
                        "api.taskkit.todoitems.delete.own"
                    }
                };

                await this._identityService.SignInUsingPinCodeAsync<UserDataDto>(
                  signInUsingPinCodeRequest, true);

                await base.NavigationService.Navigate<ToDoListViewModel>();
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e);

            await PopupNavigation.Instance.PushAsync(new ErrorPopup());
        }
        finally
        {
            base.MakeNotBusy();
        }
    }
```

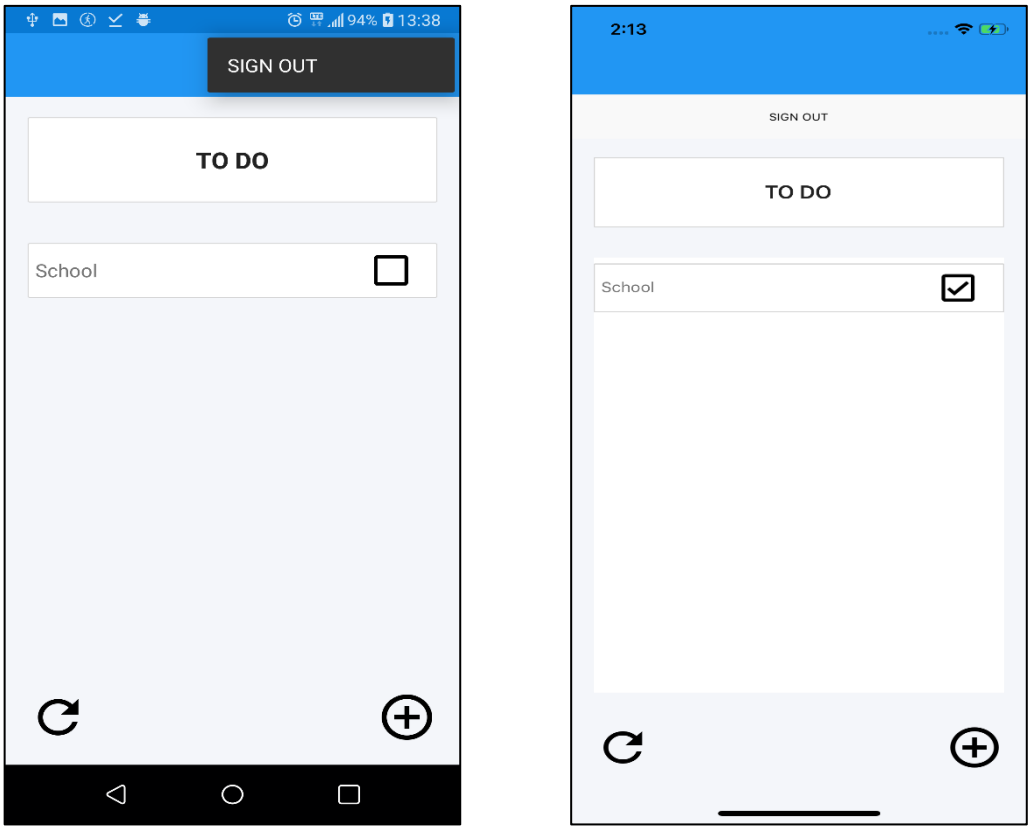


Figure 19: *ToDo List view* – one element. Own elaboration

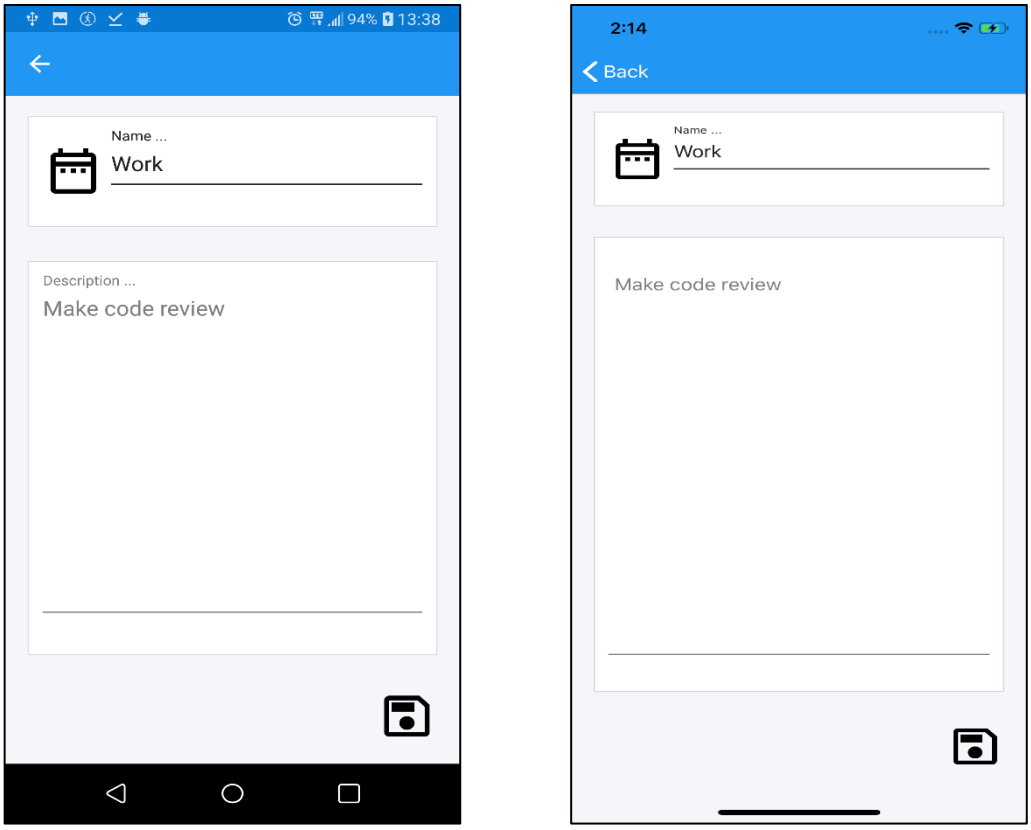


Figure 20: *ToDo Item Addition view*. Own elaboration

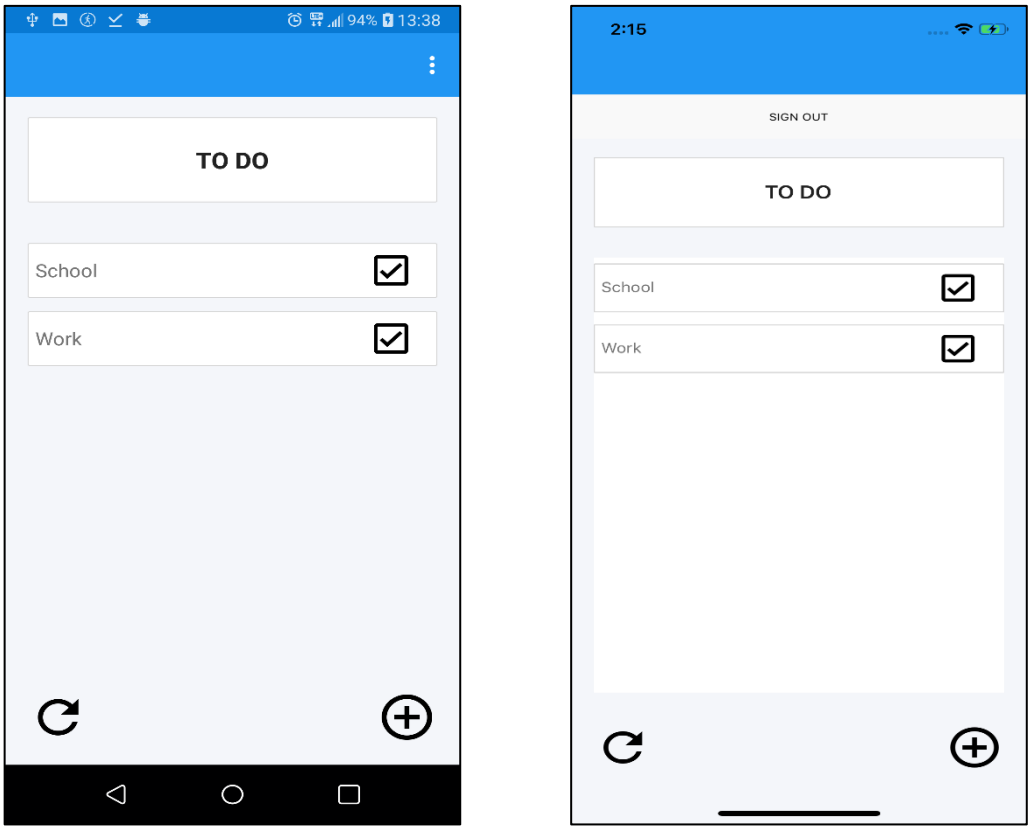


Figure 21: *ToDo List view* – two elements. Own elaboration

From the moment when a user has gone through password sign-in and enrollment processes, further sign-in operations can be done safely using only a PIN code. The PIN code login view is presented on Figure 22.

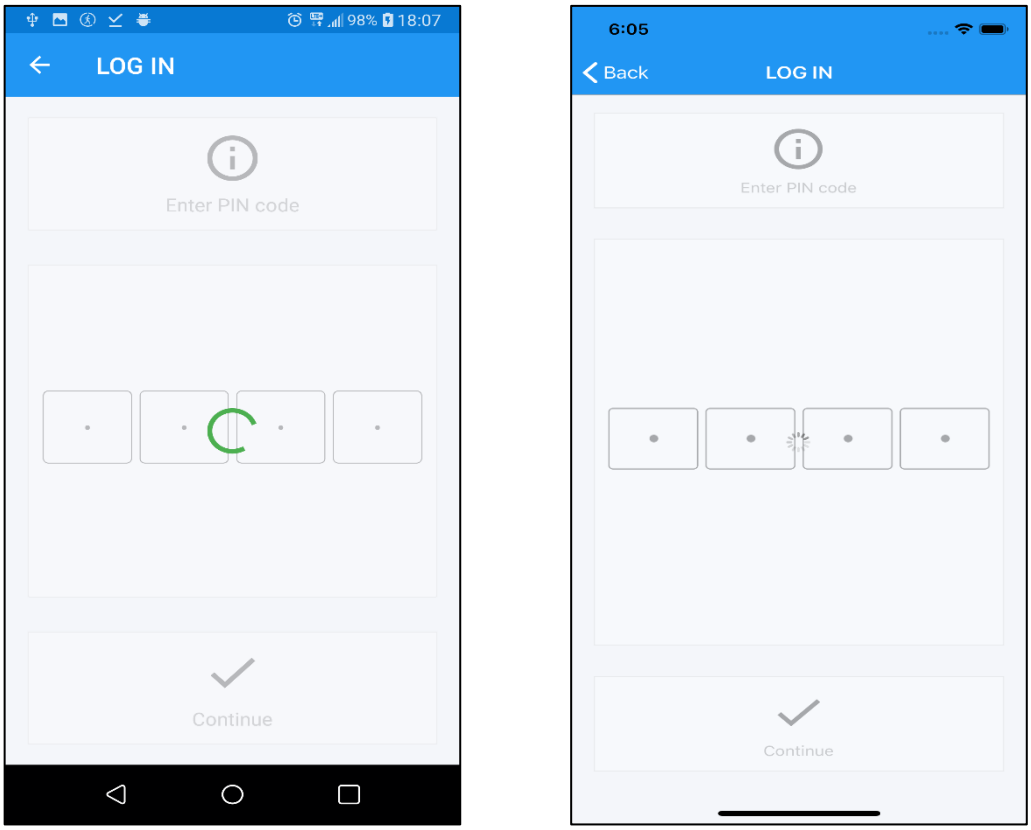


Figure 22: *Identity PIN Code Login view*. Own elaboration

10 Summary

This chapter represents a short summary of the whole thesis. It points out advantages and disadvantages of the proposed framework, shows possible directions of future supplementary works and depicts the most difficult areas which the author has gone through during his work on the solution.

10.1 Advantages and disadvantages

Building the prototype framework has shown that despite the undeniable benefits that the proposed solution brings, there is still some space for improvement as there are several areas to be dealt with to correspond to substantial needs. Below list presents the most important advantages and drawbacks noted by the author:

- Advantages
    - The server part of the framework is easy to use as it follows common .NET Core pattern to add new features to the Web API by relevant extension methods executed during its start-up.
    - The mobile part of the prototype delivers a comprehensive and configurable solution hidden by simple and meaningful methods.
    - The whole framework has been built in a plug-in manner what gives an option to customise it by developers.
    - The datastore related part of the server side framework is a component which delivers generic and abstract approach which gives an option to use any datastore.
    - The whole solution is competitive framework comparing to existing solutions and has a high potential for further development.
    - The proposed method of signing in based on user PIN code and TOTP is a proposal of logic which might become a new standard in the future.
- Disadvantages
    - Mobile part of the framework does not handle default Oauth 2.0 grants other than Password Credentials.
    - Logic related to certificate pinning has no implementation – only the concept.
    - The datastore related part of the server side framework has been tested only with the MSSQL databases and its usage with other engines may cause problems.
    - The logic responsible for retrieval of the certificates support only two approaches – file and *X509CertificateStore* based.
    - The whole framework is available only for .NET platform.

10.2 Direction of development

The framework and exemplary system have shown the potential of the solution and the final product that it can turn into, subject to further works required to make it complete and mature software. The list of areas which could be addressed by future implementation is as follows:

- **Authorization Code grant** – the mobile part of the framework should be extended to allow usage of the Authorization Code grant [3] as the alternative to the Password Credentials one. This option should be used in case when the client secret is not obfuscated therefore the application becomes a public client. A recommendation for such scenario is usage of the mentioned Oauth 2.0 grant as the sign-in session is performed outside of the application.
- **Generic sever side datastore framework** – the server side framework part which is responsible for communication with database should be tested with other datastores than the MSSQL.

Depending on the chosen database engine, many facets may impact the framework logic, for example not all relational databases support transactions or eager loading of the child entities. Also using no-SQL databases introduces necessity of relevant changes. The author also considers making a separate framework out of it. It is caused by the fact that generic and standardised approach for communication with any type of the datastore is in author's opinion something rare and often desired in enterprise scenarios.

- **Join the community** – as the MMAS framework is directly based on Identity Server 4 framework it could be a part of it in the future. Such approach should provide many benefits as the MMAS framework is put at risk of potential breaking changes in Identity Server 4 code. Both frameworks joined into one may deliver a better value for developers using it, thanks to usage of the shared code base, what should minimise the risk of the potential bugs.
- **Other languages** – the whole framework was built on top of the .NET platform as it is author's main course technology. What is obvious, .NET based frameworks and applications do not have a monopoly on the market, thus there is a big space to conduct appropriate research and build similar framework using other languages like for example Java.

10.3 Lessons learned

Building mobile applications in a secure manner requires a comprehensive knowledge about a lot of facets. Some of them are common however most of them require some level of advanced expertise. In the author's opinion, collecting the knowledge required to build at least the concept of the proposed software was a difficult task as the amount of documentation and books which required analysis was significant. The author also believes that areas addressed by this thesis are not the only ones and there is still a place for further analysis and enhancements to the solution.

This paper as well as the proposed framework represents an attempt to deliver a simple and a comprehensive solution for described problems. The lion's share of the working assumptions has been implemented successfully and therethrough the MMAS framework, even though in its initial form, can deliver a new and a meaningful value for the development community.

Bibliography

[1]. *Enterprise Application Patterns using Xamarin.Forms* (access on 2019-03-23), https://docs.microsoft.com/en-GB/xamarin/xamarin-forms/enterprise-application-patterns

[2]. *OpenID Connect specification* (access on 2019-03-23), https://openid.net/developers/specs

[3]. *OAuth 2.0 specification* (access on 2019-03-23), https://oauth.net/2

[4]. *OWASP Certificate and Public Key Pinning specification* (access on 2019-03-24), https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

[5]. Aaron Parecki, *OAuth 2.0 Simplified.* lulu.com, ISBN-13: 978-1387130108

[6]. *TOTP specification* (access on 2019-04-06), https://tools.ietf.org/html/rfc6238

[7]. *Identity Model library specification* (access on 2019-04-06), https://identitymodel.readthedocs.io/en/latest

[8]. *Identity Server 4 framework specification* (access on 2019-04-06), http://docs.identityserver.io/en/latest

[9]. *Keycloak product specification* (access on 2019-04-07), https://www.keycloak.org

[10]. *Auth0 platform specification* (access on 2019-04-07), https://auth0.com

[11]. *Red Hat official website* (access on 2019-04-07), https://www.redhat.com/en

[12]. *WSO2 Identity Server specification* (access on 2019-04-07), https://wso2.com/identity-and-access-management

[13]. *.NET architecture* (access on 2019-04-20), https://docs.microsoft.com/en-GB/dotnet/standard/library-guidance/cross-platform-targeting

[14]. *C#* (access on 2019-04-20), https://docs.microsoft.com/en-GB/dotnet/csharp/tour-of-csharp

[15]. *Stackoverflow developer survey* (access on 2019-05-04), https://insights.stackoverflow.com/survey/2018/#technology-most-popular-development-environments

[16]. *REST Client extension for Visual Studio Code* (access on 2019-05-04), https://marketplace.visualstudio.com/items?itemName=humao.rest-client

[17]. *Visual Studio - Code Map reference* (access on 2019-05-04), https://docs.microsoft.com/en-GB/visualstudio/modeling/map-dependencies-across-your-solutions?view=vs-2019

[18]. *PlantUML tool* (access on 2019-05-04), http://plantuml.com

[19]. *ASP.NET Core application start-up* (access on 2019-05-04), https://docs.microsoft.com/en-GB/aspnet/core/fundamentals/startup

[20]. *ASP.NET Core controller* (access on 2019-05-04), https://docs.microsoft.com/en-GB/aspnet/core/tutorials/first-mvc-app/adding-controller

[21]. *Identity Server 4 docs – extension grant* (access on 2019-05-04), http://docs.identityserver.io/en/latest/topics/extension_grants.html

[22]. *Microsoft SQL Server official site* (access on 2019-05-04), https://www.microsoft.com/en-GB/sql-server

[23]. *MvvmCross reference* (access on 2019-05-04),

https://www.mvvmcross.com
[24]. *SQLLite-net project site* (access on 2019-05-11),
https://github.com/praeclarum/sqlite-net
[25]. *SQLCipher project site* (access on 2019-05-12),
https://www.zetetic.net/sqlcipher
[26]. *Donald E. Knuth, The art of computer programming, 3rd edition, section 3.6*.
Addison Wesley Longman, 1998, ISBN 0-201-89684-2
[27]. *MITM – blog post* (access on 2019-05-25),
https://doubleoctopus.com/blog/the-ultimate-guide-to-man-in-the-middle-mitm-attacks-and-how-to-prevent-them
[28]. *Finn Brunton, Helen Nissenbaum, Obfuscation, A User's Guide for Privacy and Protest*.
The MIT Press, 2015, ISBN 978-0-262-02973-5

List of figures:

List of listings:

List of tables: