BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

E-me, the secure document-handling application

Abstract

EZ AZ OLDAL NEM RÉSZE A DOLGOZATNAK!

Ezt az angol kivonatot külön lapra kell nyomtatni és alá kell írni!

A DOLGOZATTAL EGYÜTT KELL BEADNI!

Kötelező befejezés:

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

2021 BALÁZS MÁRK

ADVISOR:
ASSIST PROF. DR. KOLUMBÁN SÁNDOR

BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

E-me, the secure document-handling application

ADVISOR:
ASSIST PROF. DR. KOLUMBÁN SÁNDOR

STUDENT:
BALÁZS MÁRK

2021

UNIVERSITATEA BABEŞ–BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licenţă

E-me, aplicația de gestionare securizată a documentelor

CONDUCĂTOR ŞTIINŢIFIC:
LECTOR DR. KOLUMBÁN SÁNDOR

ABSOLVENT:
BALÁZS MÁRK

2021

BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Szakdolgozat

E-me, a biztonságos dokumentum-kezelő alkalmazás

TÉMAVEZETŐ:

DR. KOLUMBÁN SÁNDOR,
EGYETEMI ADJUNKTUS

SZERZŐ:

BALÁZS MÁRK

2021

Contents

1. Chapter

Introduction

1.1 About E-me

A brief introduction of the application, 1-2 pages.

- general introduction
- why somebody would use this app
- the main features/selling points of the app

1.2 Similar applications in the field

A list of similar applications, their advantages and disadvantages, comparisons, 2-3 pages.

1. Google Docs
   - create and edit documents
   - sync between multiple devices
   - view PDF docs/presentations
   - upload and manage files
2. Documents to Go
   - edit/view/create word, excel, PowerPoint docs
   - supports password protection
   - Google Docs support
   - bi-directional sync
3. SecureSafe

   - secure file and data storage
   - double encryption
   - secure AES-256 and RSA-2048 encryption
   - https
   - MFA with SMS
   - send files up to 2GB to recipients

4. Quick Office Pro
   - create/edit/share Microsoft Office files
   - offline file access

1.3 Comparison

1.3.1 Similarities

- Similarly to the **SecureSafe** app, E-me uses AES-256 symmetric encryption standard to securely store and transfer documents.
- E-me allows users to upload their PDF documents.
- Users have quick and secure access to their data and files.

1.3.2 Differences

- E-me only supports PDF documents.
- E-me uses End-to-End Encryption over HTTPS to communicate with the clients.
- Users are able to **generate** their PDF docs using predefined templates filled out with their personal data.
- All PDF documents **(generated or uploaded)** will be verified for authenticity by the system administrators (later government) and will receive a digital signature to mark their authenticity.
- Authorities can request access to users' documents in order to verify their identity or other personal information (this access is temporary).

1.4 Summary

Describes the structure of the following document, 1 page.

2. Chapter

User documentation

Summary: *In this chapter E-me is described from a user point of view.*

The building blocks of E-me are pages. The primary role of these pages is allowing the user to manage their PDF documents that were created through the application. This management includes the ability to request a new document, view already owned documents, make modifications to them, delete the ones that are obsolete or contain incorrect data, or even share them with other devices using a QR code.

The secondary role is collecting the data from the user that is necessary for generating their documents. This data is then encrypted, processed and later on added to the documents that the user requests.

2.1 Introductory Pages

For an unauthenticated user there are 3 pages available in the application: Greeting, Login and Registration. Upon opening the application, the user is confronted with the Greeting page which allows them to navigate to the Login and Registration pages using the dedicated buttons.

The Registration page consists of five text fields. Each of these fields are required in order to create a new user, however only two of them are needed for authentication: Login name and Password. The restrictions for these fields are the following:

- The Email and Login Name fields should be unique.
- The Password and Confirm password fields should be identical.
- The Email field should be a valid email address.

Upon successful registration, the application navigates the user to the Login Page in order for them to authenticate themselves using the Login Name and Password they entered. If the authentication was successful, the main shell appears where three tabs can be seen: My Documents, Request Document and Personal Info.

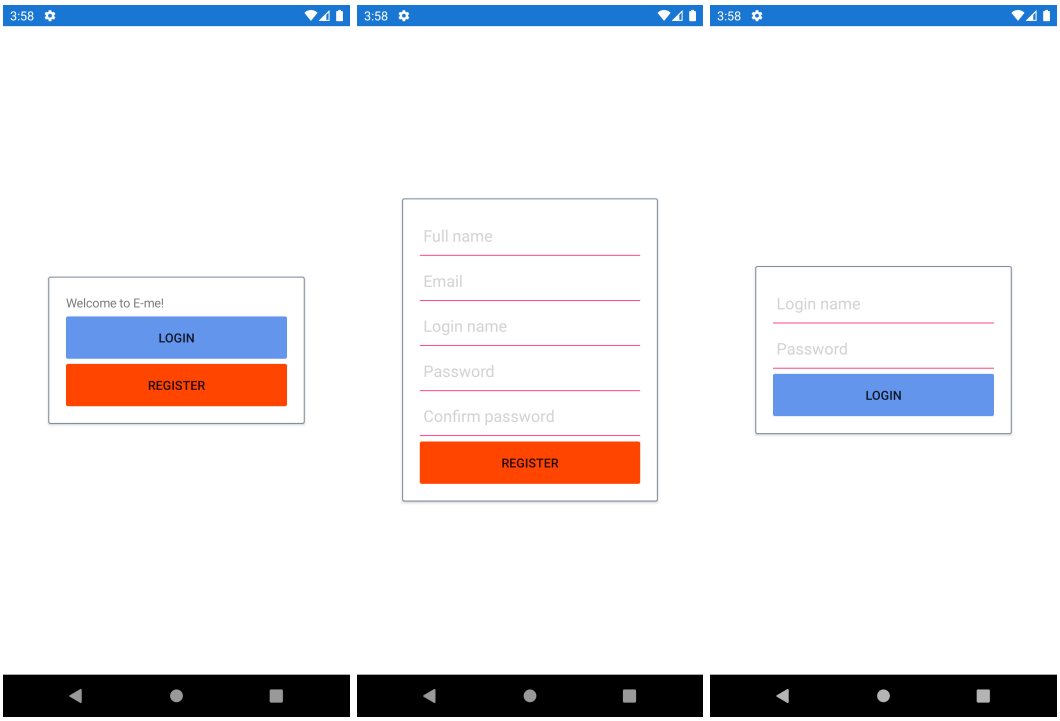

Figure 2.1: Introductory pages (Greeting, Registration, Login)

2.2 My Documents

The My Documents page consists of two main parts: the list of documents and the "Scan QR code" button. The list of documents allows the user to visualize what types of documents they own. On each document two actions can be performed: Share and Remove.

Tapping the Remove button irreversibly deletes the document from the list. After a successful removal the user is able to request a new document of the same type. If the document template was modified since the previous request or the user modified their personal information, the resulting document may be different than the previously deleted one.

The Share button allows the user to safely transfer their selected document to a different device. Upon tapping the button, the application generates a unique code for the document which is then displayed on the screen via a QR code.

The code can be read using the "Scan QR code" button. Tapping this button will attempt to open the device's main camera in order to scan the code. If this feature is accessed for the first time, a prompt appears asking for the user's permission to use the camera. If the permission is granted, the application will open the camera app. Upon successfully reading a QR code generated by E-me, the selected document will appear on the screen (see Document Viewer).

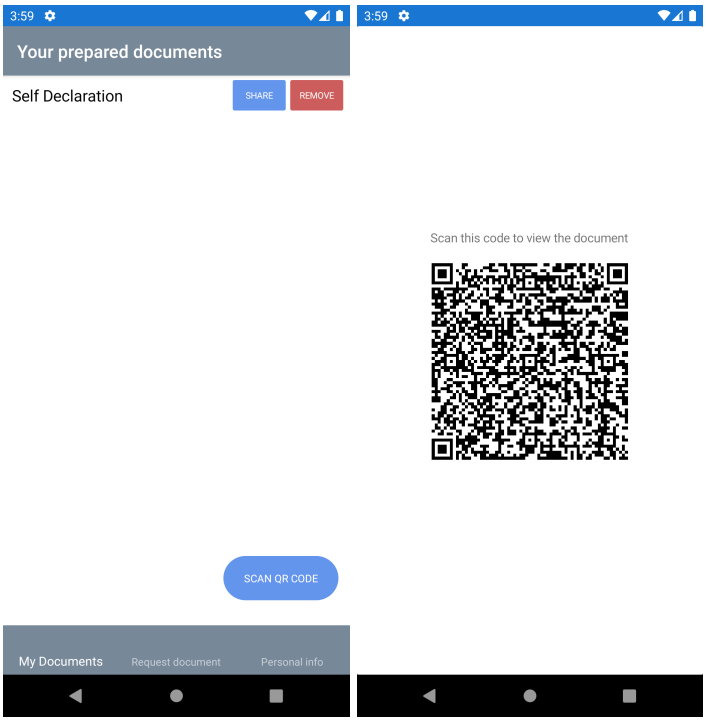

Figure 2.2: Documents and Share Document pages

2.3 Requesting a Document

The Request Document page consists of a list of document types that can be acquired by the user. The list only contains types that are not yet owned by the corresponding user. If the user acquires one of these document types, it will disappear from the list (it will be listed on the My Documents page instead).

Combined with the My Documents page, these two lists contain every document type that can be managed through the application.

The user can request a document by simply tapping on an item from the list. The requested document will be generated and automatically displayed on the screen (see Documents Page).

2.4 Personal Information

This page contains several fields containing the personal details of the user. Neither of these fields are required, however if no information is provided about the user, the application will not be able to automatically fill the requested documents. The information provided by the user on this page will be stored in an encrypted form.

Upon requesting a document, E-me attempts to match the fields of the document with the information of the user and fill them out respectively. Fields that require information that is not

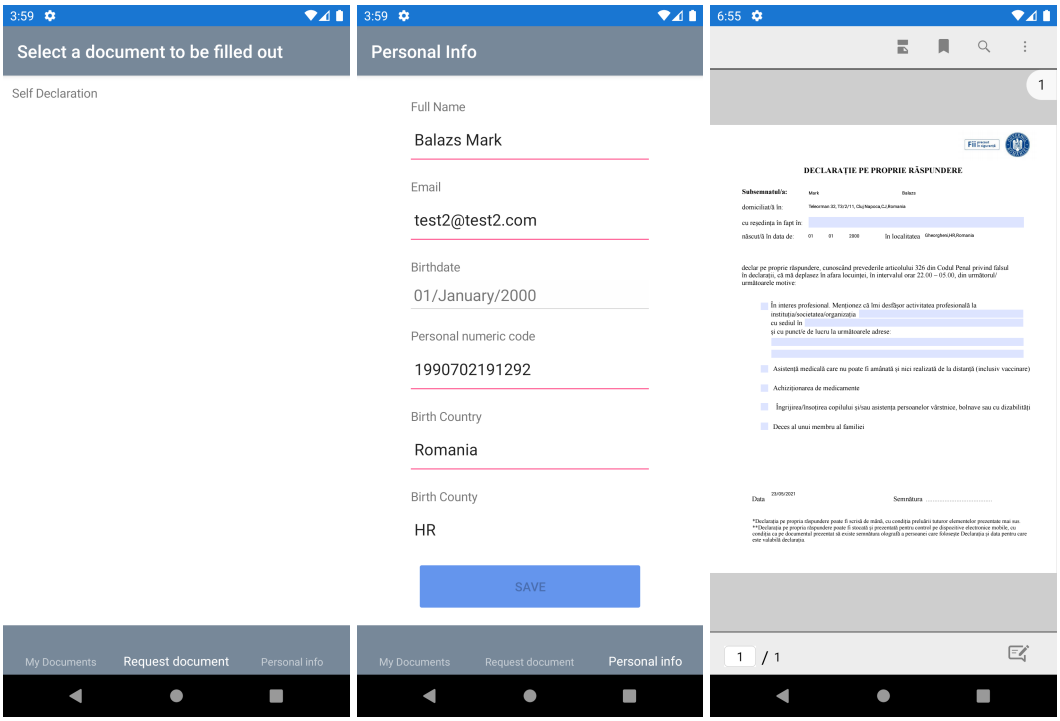

Figure 2.3: Request Document, Personal Information and Document Viewer pages

provided by the user will be left blank and can be filled manually on the Document Viewer.

2.5 Document Viewer

On this page a PDF Viewer can be seen which allows the user to inspect or even edit their documents. This viewer supports text searching, bookmarking, zooming, printing and/or saving a PDF to the local storage of the mobile device. The contents of the PDF are generated by the application, however if the document contains fields that could not be matched with any user data, the user is able to fill them out manually. This page can be closed by pressing the Back button.

Figure 2.3 contains an example of a Romanian COVID-19 self-declaration. The document consists of multiple input types, including checkboxes, dates and text fields. The majority of these inputs refer to general information about the user (name, date of birth etc.) which can be filled out automatically by the application, however some fields require time- or location-dependent data, which is not stored in E-me. In this case, these fields are left blank so that the user is able to complete them manually.

3. Chapter

Technical details

3.1 Architecture

E-me follows a commonly used N-tier architecture with three main parts: data, application (backend) and presentation (frontend) layers. Each of these tiers can be broken down into layers that are defined by their responsibilities within the application. This tier-based architectural approach adds modularity to the application which results in a low cost of change when compared to a single-tier structure.

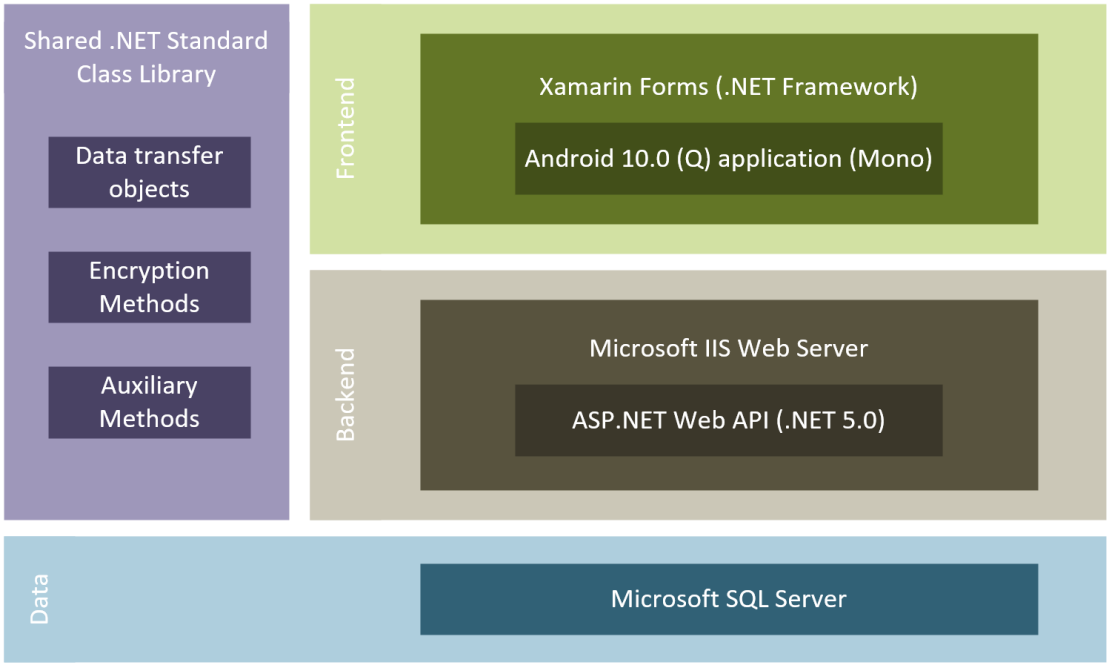

Figure 3.1: General architecture

Along with the low cost of change, the independence of the layers allows for efficient future expansion of the application by means of multiple frontend platforms (ex. web applications, desktop applications), cloud storage/services and additional Web API's that can easily be integrated into the existing application. This, combined with the high compatibility of the .NET

5.0, provides a high level of scalability and maintainability for the application.

The application and presentation layers share a common class library that contains communication-related models and auxiliary methods. This library allows both layers to benefit from the cross-platform nature of .NET 5.0, speeding up the process of development and ensuring there are no discrepancies betweeen the layers in terms of encryption and communication.

3.1.1 Application layer / Backend

The architecture of the application layer has a similar design approach to the general architecture. Consisting of 4 layers, the backend follows the single-responsibility principle in it's core. Because of it's vertical structure, each layer is dependent on one single layer that is directly below it, providing a high level of maintainability.

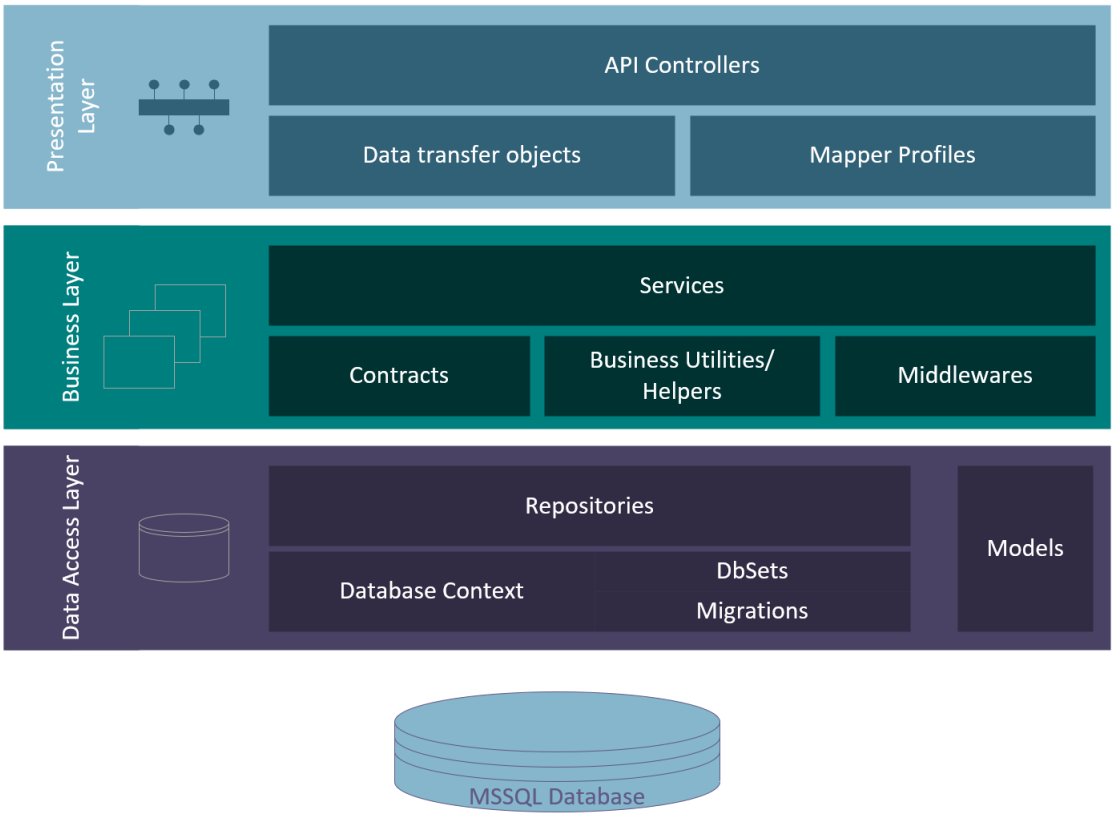

Figure 3.2: Backend architecture

Due to this abstract layout, each layer can independently be replaced or updated. This aspect enables the utilization of external and/or third-party services and components without damaging the integrity of the application.

The API structure of the presentation layer allows for a wide variety of applications or even services in which the backend can be used: desktop applications, WPF, web servers and more.

This characteristic has major role in preserving the flexibility of the application layer.

Another component being responsible for the independece and reusability of the backend is the Data Access Layer. The role of this layer is to provide the data requested by the Business Layer. By reason of abstraction, this layer is independent of the technology of the data source. This enables the utilization of a wide range of data sources, including relational (SQL) and non-relational (NoSQL) databases, cloud services and/or APIs.

Entity relations in the database layer

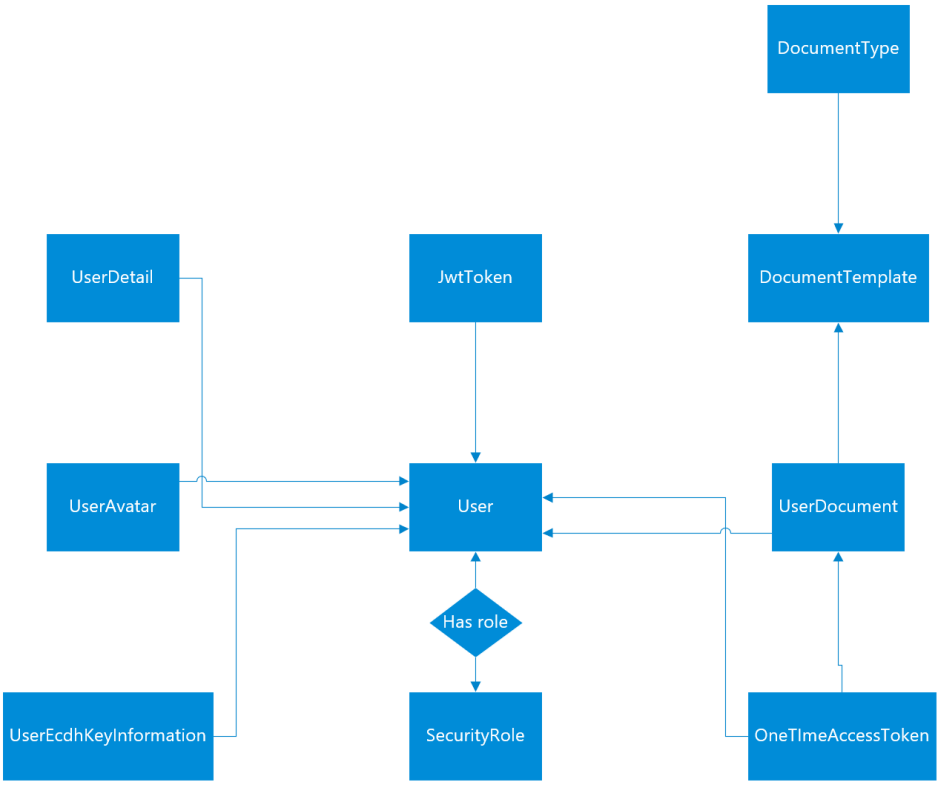

Figure 3.3: Entity-relationship diagram

3.1.2 Presentation layer / Frontend

The frontend of the application consists of three major layers that separate the user interface from the business logic and the data sources. This separation allows for easy horizontal expansion and quick feature development.

The Presentation or UI Layer contains the visuals of the application which are separated into independent pages, however it is also responsible for receiving user events and connecting them with the underlying services. Each page contains it's own event listeners and separate View Model that is responsible for making use of the Business Layer, which has a similar structure and role to the backend's Business Layer: data processing and calculations.

The main components of the Data Layer are Data Stores. These stores are functionally similar to the repositories found in the Backend of the application, although here the data is retrieved using the endpoints of the backend.

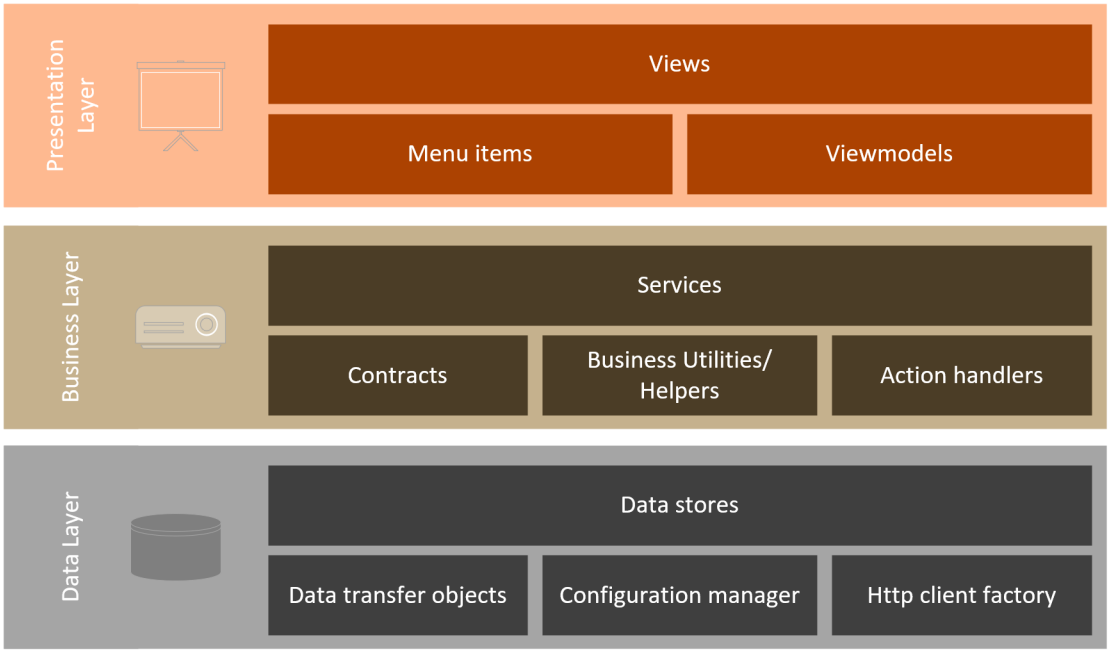

Figure 3.4: Frontend architecture

3.2 Technologies

E-me makes great use of the cross-platform nature of .NET, meaning both the frontend and the backend of the application are implemented using the same programming language: C# 9. The flexibility that is provided by the language and the wide variety of frameworks that make use of it allows for a cleaner codebase and a simpler project structure when compared to a multilingual application design.

3.2.1 Backend

In terms of frameworks, the backend relies on .NET 5.0. Being the next major release of .NET Core following 3.1, this open-source cross-platform software framework unifies a wide variety of frameworks .NET has to offer, including Windows Forms, ASP.NET Core, Azure and more. This unification made the framework a perfect candidate for the backend of an application like E-me, since it can be used on a wide variety of platforms and ensures long-term support.

E-me uses Internet Information Services (Microsoft IIS), which is an extensible web server software, as it's backend server. It allows the application to communicate through HTTPS and

provides it's separate development-time SSL certificate in order to secure the transport layer. With the *Development-time IIS support* feature, the web server allows *hot reloading*, which speeds up the development and debugging processes.

The main constituent of the backend itself is the ASP.NET Core web framework. Using it's built-in IoC, this framework allows for a high level of abstraction in order to build well-structured enterprise level applications using dependency injection. This framework is responsible for registering and configuring any other dependencies that may appear in the application backend.

Alongside with ASP.NET Core, one of the main components of the backend is the Entity Framework Core, which is the base of the *Data Access Layer*. In terms of building and maintaining the data access layer, E-me uses the code-first approach through *EF Migrations*. This approach provides flexibility in case of a potential database change and/or creation, but more importantly it allows the entire structure of the entities to be managed based on the models and datasets present in the code.

E-me heavily relies on encryption and security, which demands the utilization of advanced cryptographic libraries. For this purpose, the application uses the *Windows CNG (Cryptographic Next Generation) API*, which enables exchanging documents and other data in a secure environment. In terms of cryptography, this API provides a full implementation of the *Elliptic-curve Diffie-Hellman* key agreement protocol, alongside with many other cryptographic protocols and algorithms such as RSA, AES (Advanced Encryption Standard) or DSA (Digital Signature Algorithm) and hash functions like SHA-2 or MD5.

The documents handled by the application are created and managed by the *Telerik Document Processing Core* framework, which provides flexibility and efficiency in terms of document management. It enables processing the most commonly used text, PDF and spreadsheet file formats, allowing the application to create, import and export documents without relying on other external dependencies.

3.2.2 Frontend

Being a .NET-based application, E-me uses Microsoft's open-source mobile UI framework, *Xamarin Forms* as a base for it's frontend. This library allows developing native mobile applications on multiple platforms, such as iOS, Android or UWP using *.NET Standard*. The multi-platform capability allows for a shared codebase and logic across all applications, which speed up the development process immensely. Xamarin uses an XML-based interface designer in order to create the visuals for the application. Alongside with the commonly used Xamarin UI components, E-me uses the *Telerik UI for Xamarin, GoogleVision API and Syncfusion* UI libraries for additional components and controls in it's user interface.

Although the frameworks used by the different layers of the application vary, these can have a shared codebase in the form of a *.NET Standard class library*, which can be used by both the backend and the frontend, allowing for closely integrated layers.

3.3 Implementation

Being cross-platform and having a wide range of possibilities in terms of future expansion and upgrades, E-me focuses heavily on scalability and maintainability. In order to achieve these characteristics, multiple factors have to be considered during development, such as project structure, dependencies and code reusability. For this purpose, both the backend and the frontend of the application relies on *Dependency Injection*. This technique allows for a service-based structure within the application and it is an elegant way of implementing loosely coupled classes.

3.3.1 Backend

The backend of E-me is separated into 5 projects that are defined by their role within the application. These projects are targeting the *.NET 5.0* framework, one exception being the *Shared* project, which is used as the common codebase for the backend and the frontend. The majority of the projects are *Class Libraries*, which are collections of pre-coded object-oriented templates and classes. These libraries revolve around the main entry-point of the application, the *MVC project*.

The MVC project

As the main entry-point of the application backend, this project is responsible for configuring and registering all dependencies and services E-me requires and utilizes. This configuration of the *IoC (inversion of control)* and the registration of dependencies take place in the *Startup* class. This class is registered to the *Host Builder* as soon as the application starts.

```
1  public static IHostBuilder CreateHostBuilder(string[] args)
   {
         return Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
6                       webBuilder.UseStartup<Startup>();
                });
   }
```

Representing the presentation layer, this project defines the *API Controllers* which are representing the endpoints of the backend. These controllers are the connection points through which the backend is able to communicate with other layers of the application. The endpoints of

these controllers follow the same route structure: *[base-address]/api/[controller-name]/[action-name]*, for example: *https://e-me.Mvc/api/auth/login*. All endpoints of the application require *HTTPS* to be accessed and send response in *JSON* format to ensure that the receiving party is able to interpret the message.

One of the most important of these controllers is the *AuthController*, which is responsible for registering, authenticating and de-authenticating the users of the application. This is the only controller that can be accessed by an unauthenticated user, since the rest of them handle more sensitive data.

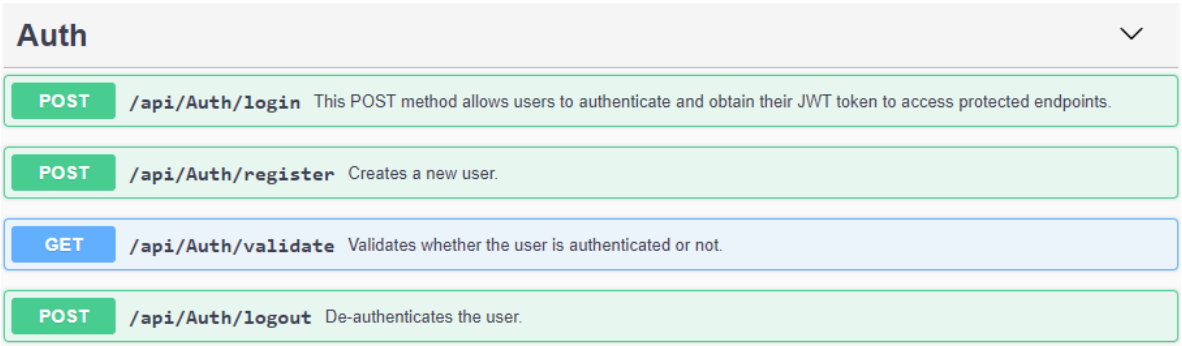

Figure 3.5: AuthController endpoints

Once a user is authenticated, they are able to access the protected endpoints. A controller's or and endpoint's protected nature is marked by the *Authorize* declarative attribute. When accessed, the endpoints marked with this attribute automatically verify the authorization header of the request and the role of the current user if necessary.

```
[Authorize]
[ApiController]
public class UserDetailsController : Controller
```

Two of the main protected controllers are the *UserDocumentsController* and the *DocumentTemplatesController*. These two combined are responsible for handling any document-related user requests, such as reading or deleting by id, getting available or owned documents etc. All of these actions contain an extra layer of validation in order to ensure that the current user is eligible for viewing, modifying or removing the specified documents or types.

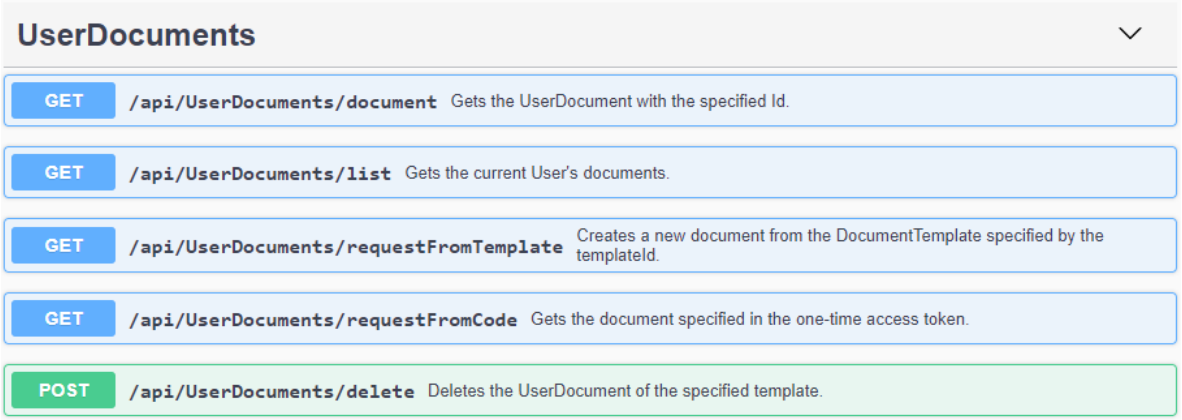

Figure 3.6: UserDocumentsController endpoints

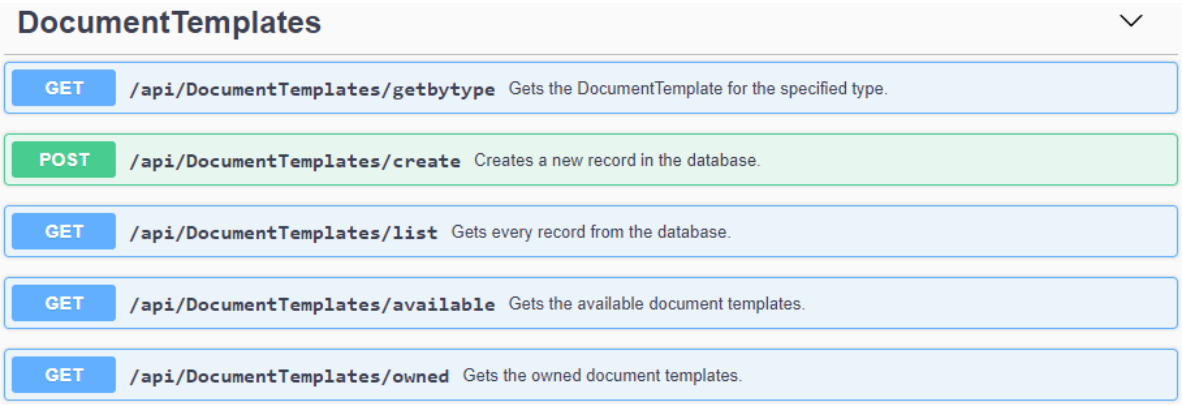

Figure 3.7: DocumentTemplatesController endpoints

Apart from these three major controllers, there are three that are significantly more compact and have much fewer endpoints: *DocumentTypesController, OneTimeAccessTokensController* and *UserDetailsController*. These three cover the non-document-related actions within the application, which include modifications of the personal information of the user, requesting a *document-sharing token* (represented as a QR code in the frontend) or listing the document types that can be managed through the application.

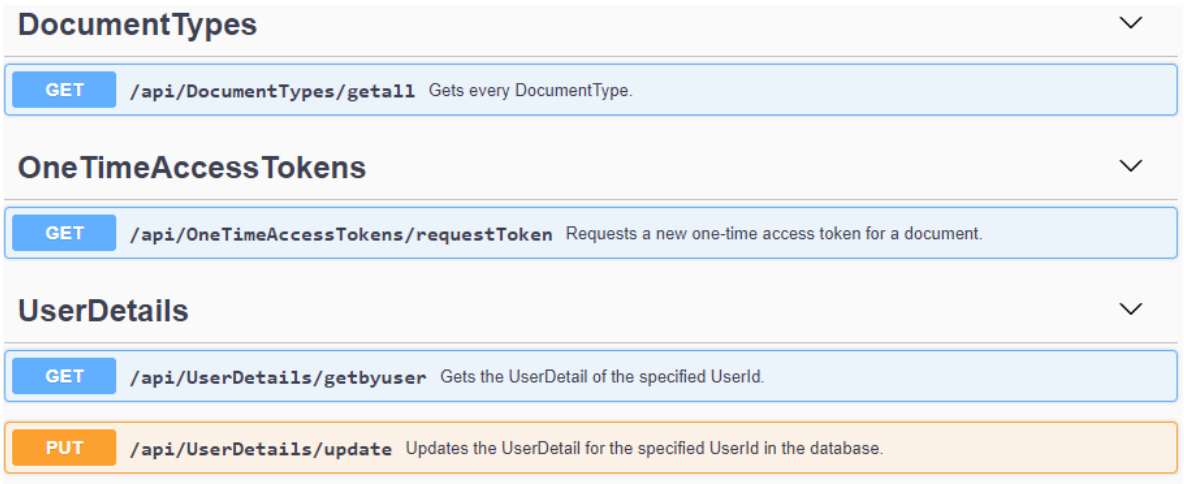

Figure 3.8: Endpoints of the minor controllers

The Business project

As the name suggests, this intermediate project represents the Business Layer of the application backend. The project itself consists of service contracts (interfaces), factories and implementations that make use of the data-access objects from the data layer. These services perform calculations, alterations and complex operations on the data provided by the underlying layer and are utilized by the above mentioned MVC project as the source of processed data. They are registered in the *Startup* class and injected in the constructors of the controllers by the *IoC container* upon accessing the endpoints of the corresponding controller:

```
public AuthController(IAuthService authService, IUserService userService)
{
    _authService = authService;
    _userService = userService;
}
```

This loose class coupling allows for multiple implementations of the services. These implementations follow the same structure (contract), but don't necessarily have matching outputs and/or dependencies. The implementation is selected either using the *factory pattern* or in the moment of registering the service to the *IoC container*. Example for a service contract:

```
public interface ITokenGeneratorService
{
    string Generate(string userName, DateTime validTo, string role);
    string GeneratePasswordResetToken(string username, DateTime validTo
        );
    string GenerateOneTimeAccessToken(Guid userDocumentId, DateTime
        validTo);
}
```

The Model project

One other major building block of the backend is the *Model* project. As the *MVC* represented the presentation layer of the application backend, the *Model* implements the data-access layer. Having said that, this project is responsible for data persistence-related operations via *Entity Framework Core*. In order to manage the data on the server, the project has to model the structure of the database in the form of classes (*Models*) and *DbSets* which represent tables and the underlying data respectively.

The framework allows for multiple directions in managing the database structure (ex. database-first or code-first), out of which E-me uses the code-first option. This kind of implementation denotes that the database is automatically generated by the framework using the pre-coded models and defined constraints. For representing various versions of the data structure that may appear during development, database migrations are used which contain the alterations that occured after the previous version of the database was generated. These migrations can be reverted and/or modified, providing flexibility when designing the data-layer and also ensuring that the structure of the database is correctly representing the model-structure from the code at all times.

The outermost layer of the Model project consists of *repositories*. These classes represent the data-access objects (*DAO*), which are used by the services in the Business Layer to retrieve raw or pre-processed data. Each data model has its own repository that allows for performing the basic operations on the corresponding table from the database. These operations include deletion, insertion and querying. Each repository is inherited from the *BaseRepository* abstract class, which implements the above mentioned basic operations:

```
public abstract class BaseRepository<TEntity> : BaseRepository,
    IBaseRepository<TEntity> where TEntity : Models.BaseModel
```

The more complex operations and non-generic queries are implemented individually in each repository depending on the needs of the corresponding service that uses the DAO. This can be seen in the following example:

```
public class OneTimeAccessTokenRepository : BaseRepository<
    OneTimeAccessToken>, IOneTimeAccessTokenRepository
{
        ...
    public async Task<OneTimeAccessToken> FindByUserDocumentIdAsync(Guid
        userDocumentId)
        {
                return await All.FirstOrDefaultAsync(p => p.UserDocumentId
                    == userDocumentId);
        }
}
public interface IOneTimeAccessTokenRepository : IBaseRepository<
    OneTimeAccessToken>
{
        Task<OneTimeAccessToken> FindByUserDocumentIdAsync(Guid
            userDocumentId);
```

```
}
```

3.3.2 Frontend

In terms of volume, the frontend of the application is substantially smaller than the backend. It consists of 3 projects, one of them being automatically generated as a result of using *Xamarin Forms*. The main entrypoint here is the *Mobile* projects which contains the majority of the frontend implementation and can be separated into 4 major building blocks: services, models, views and view models (MVVM).

The Mobile project

Being a *Xamarin Forms* project, the Mobile part of the frontend is a *.NET Standard 2.1* class library which contains the shared codebase across all the mobile platforms that are part of the application. This project serves as a template for Xamarin to generate the native code for each individual platform (Android, iOS or UWP). In case of E-me, the Android platform is targeted which denotes that the generated libraries and binaries are using *Mono runtime* in order to access all of the native Android APIs.

The setup of the project is similar to the *MVC* project from the backend of E-me, since it uses the same concepts when it comes to inversion of control and dependency injection. Similarly, a *Startup* class is used to register services, visual components (views) and view models to the *IoC container* for loose coupling.

The main component of the presentation layer is the *AppShell*. This shell is a tab-based wrapper for the views of the frontend. The shell's visibility alters depending on the currently rendered view; some views are not required to be seen among the tabs of the application shell (ex. Login view, Registration view).

The visual components/views of the frontend are refferred to as *pages*. Each page consists of two parts: an XML file representing the visuals of the page and a C# class that provides the functionality for these visuals. The various view models and services are injected into the constructors of these underlying classes:

```
public RegisterPage(RegisterViewModel registerViewModel, IUserService
    userService, IMapper mapper, INavigationService navigationService)
{
        _userService = userService;
        _mapper = mapper;
        _registerViewModel = registerViewModel;
        _navigationService = navigationService;
        InitializeComponent();
        BindingContext = registerViewModel;
}
```

The services of the frontend include *cryptographic, communicaiton and storage services*. The cryptographic service makes use of the shared key-derivation and encryption methods implemented in the *Shared* project. It is used in the decryption of documents received from the application backend and upon generating keys for the *Diffie-Hellman key exchange*(see End-to-end encryption).

Communication services are responsible for establishing connection with the backend of the application. This connection is made possible by *HttpClients* which are retrieved from a *HttpClientFactory*. The role of this factory is to create clients that already include the authorization header (if the user is authenticated) and base address of the backend server. These services create request messages based on the needs of the user and handle the interpretation of the response from the server (deserialisation).

E-me makes use of the *Android Keystore system* in order to securely store sensitive information (keys, tokens etc.) via the storage services. These services are responsable for reliable data retrieval either from the keystore or from configuration files depending on the needs of the user.

3.3.3 Use cases

The use cases in E-me can be categorised by their actors, since they don't have overlapping instances.

Unauthenticated user

As for a unauthenticated user, there are two possible actions that can be perfomed: Login and Registration. The detailed description of the login process can be found in the Security section.

The process of registration starts at the *Registration page* on the frontend. Upon submitting the registration form, a *UserRegistrationDto* object is created containing all the necessary information for creating a new user. After a successful backend validation via the *UserService*, a new record in the User table is created, followed by new records in the *UserSecurityRole* and *UserDetail* tables. If the creation of the new records was successful, the backend server sends an adequate response (200 OK) in order to signal the client about the successful registration.

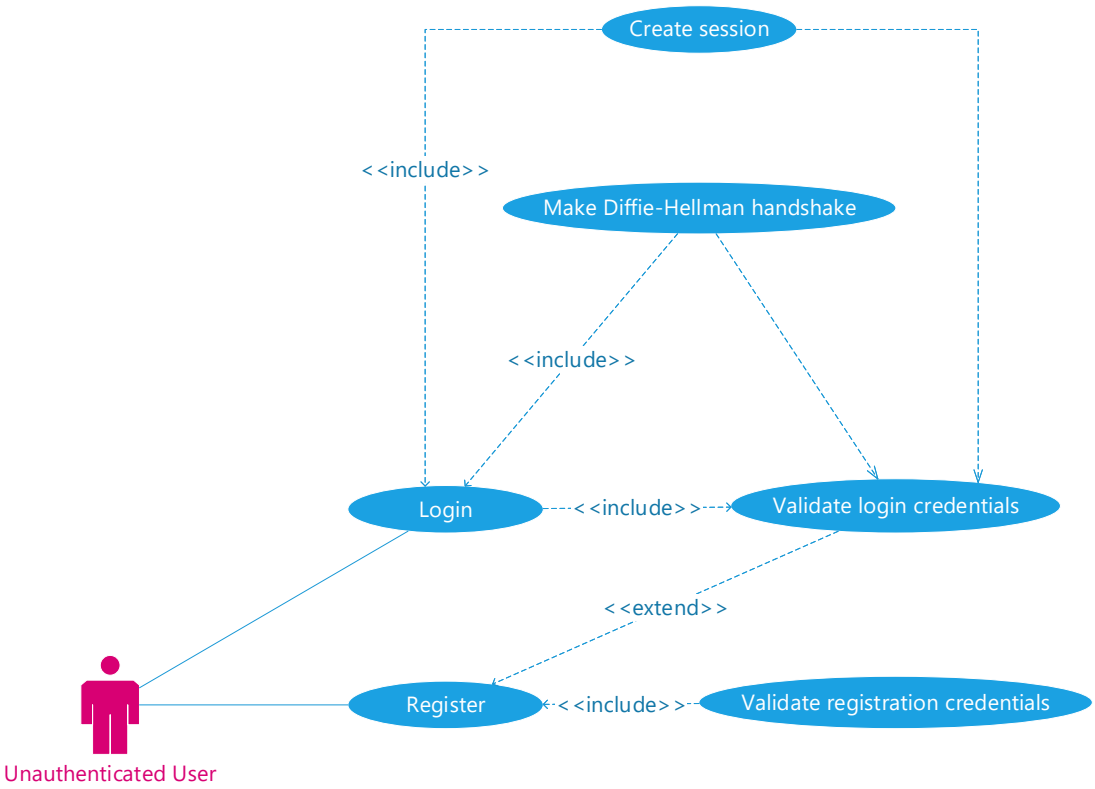

Figure 3.9: Use-case diagram (unauthenticated user)

Authenticated user

Following a successful login, a user is able to perform their document-related actions, which include viewing, sharing, deleting and/or requesting documents, and also have access to their personal information.

One of the main features of E-me is the automatic document creation using the personal details of the user. The process of requesting a document starts at selecting a document type (template) from the list of available documents. Using the *DocumentService*, the frontend proceeds to send the id (GUID) of the template to the *RequestFromTemplate* endpoint of the server. Upon receiving a valid template ID, the server loads the reuqested template document from the database. Using the *Telerik Document Processing* library, the *DocumentProcessorService* identifies the form fields of the corresponding document and looks for matching fields in the personal details of the user. Upon mapping the fields, the service proceeds to fill the ones that have matching information about the user.

```
public UserDocument GetDocumentFromTemplate(DocumentTemplate
    documentTemplate, UserDetail userDetail)
{
        var userData = userDetail.MapToDictionary();
        var radDocumentTemplate = DocumentProcessing.
            GetFixedDocumentFromBytes(documentTemplate.File);
        DocumentProcessing.FillDocumentFormFieldsByFieldNames(
            radDocumentTemplate, userData);
        var filledDocument = DocumentProcessing.GetBytesFromFixedDocument(
            radDocumentTemplate);
        return filledDocument;
}
```

Upon successfully filling the forms of the document, the newly created instance is saved in the database. After successfully creating the necessary instances in the database, the server retrieves the encryption key information about the user from the database to encrypt and hash the newly created document. If the encryption was successful, a *UserDocumentDto* is created which contains the encrypted file in *base64* format and the hash of the file to ensure data integrity and authenticity.

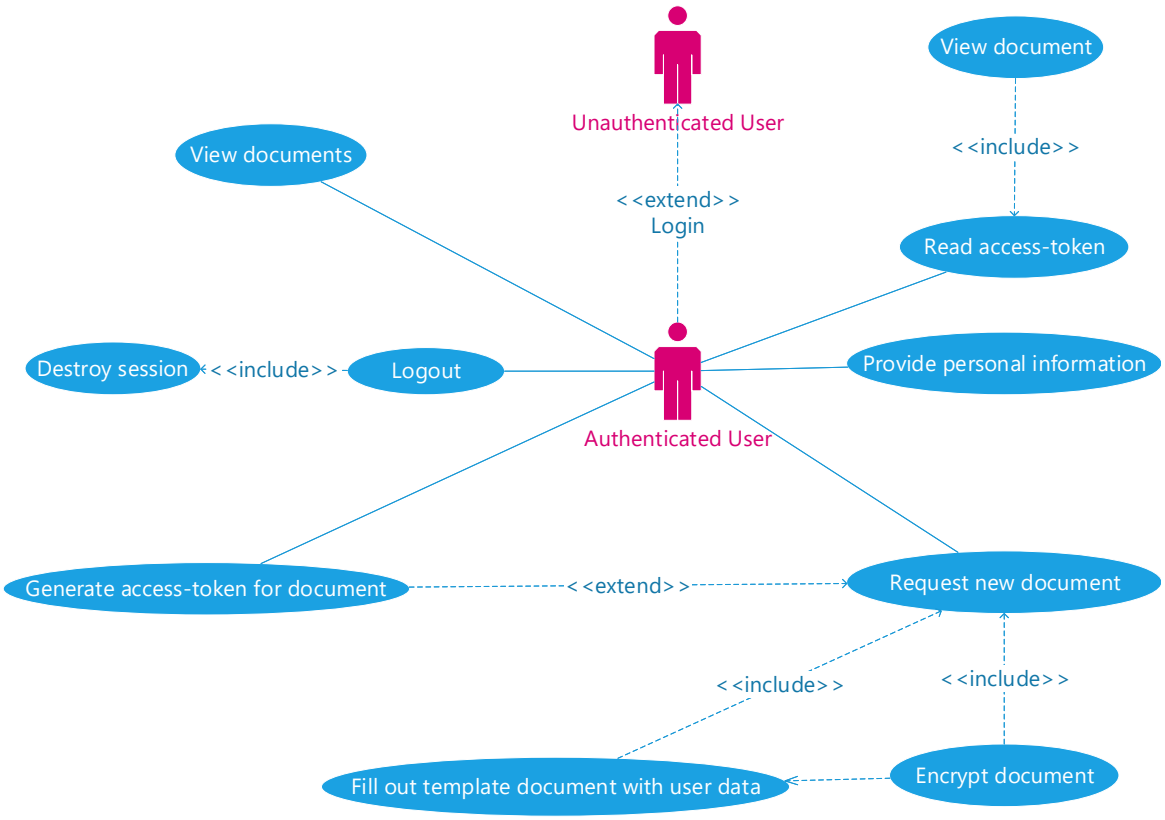

Figure 3.10: Use-case diagram (authenticated user)

After receiving the DTO from the server, the frontend client validates the hash of the document and decrypts it using the shared encryption key stored in the *Android Keystore*. If the

decryption was successful, the client presents the PDF document to the user via the *Syncfusion PDF Viewer*.

When sharing a document, the document ID is sent to the server by the *DocumentService* from the frontend. Upon receiving a correct document ID and performing validations on the user's eligibility, the backend generates a token containing information about the document which is available for *one hour* after being generated. After this token is received on the client side, it is represented as a QR code.

When reading the QR code using the built-in scanner, the mobile client retrieves the above mentioned token and sends it to the backend. In case of a valid token, the server perform an encryption and hashing of the corresponding document identical to the one performed when creating a new one. After the client receives the encrypted document, it performs the same actions as mentioned above to present the document.

3.4 Security

As a document-handling application, E-me relies heavily on secure communication, data storage and encryption. In order to achieve the required level of protection, the application uses a handful of libraries that target various layers within the backend and the frontend.

3.4.1 Data-layer security

First and foremost, E-me utilizes *Entity Framework Core Data Encryption*, which is a plugin for the above mentioned *Entity Framework Core*, to add support of encrypted fields using built-in or custom encryption providers. This library ensures the protected storage of the data via encryption. This way the data stored in the database cannot be interpreted by other parties, not even in a case of a security breach or database failure. In the case of E-me, a built-in encryption provider is used which enables *AES-256* encryption configured with a CBC (cipher-block chaining) cipher. The algorithm uses a secret key stored in the configuration of the application.

```
private AesProvider CreateAesProvider()
{
    var key = Encoding.ASCII.GetBytes(_authSettings.SecretKey);
    var provider = new AesProvider(key, CipherMode.CBC,
        PaddingMode.Zeros);
    return provider;
}
```

The data encryption library allows for independent protection of model attributes, providing flexibility for the developers in selecting sensitive fields that require encryption. This selection can greatly reduce the time consumption of the encryption and is preformed via the *Encrypted* field attribute:

```
[Encrypted]
public string FullName { get; set; }
```

3.4.2 End-to-end encryption (E2EE)

The second layer of security is located in the *Business Layer* of the application backend. One of the main features of E-me is the end-to-end encrypted (E2EE) document transfer which is achieved by this second layer. In order to successfully perform an encryption on the backend followed by a decryption on the frontend, the two sides must complete a secure key exchange beforehand. This exchange occurs upon a user's successful login to the system.

Upon receiving the user's credentials, a key pair (private and public keys) is generated on the frontend of the application. The key generation and derivation takes place in the *EcdhKeyStore* class which uses a password-based key derivation functon (*PBKDF2*) with a *SHA512* hash function as the private key generator and *X25519 Elliptic Curve Diffie-Hellman* to derive a public key from the generated private key.

```
public EcdhKeyStore()
{
        using var rngCsp = new RNGCryptoServiceProvider();
        var salt = new byte[SaltSize];
        var rawKey = new byte[RawKeySize];
        rngCsp.GetBytes(salt);
        rngCsp.GetBytes(rawKey);
        PrivateKey = new Rfc2898DeriveBytes(rawKey, salt, Iterations,
            HashAlgorithmName.SHA512).GetBytes(KeySize);
        PublicKey = Curve25519.GetPublicKey(PrivateKey);
}
```

The public part of the key pair is included in the login request in order to be processed on the backend server. Upon a successful verification of the login credentials, the backend's authentication service generates it's own private and public key pairs. These — combined with the client's public key — enable the derivation of a third key (shared secret) and a shared *AES Initialization Vector (IV)* which can be used for encryption and hashing in the future. This key derivation process is followed by the storage of the keys and hash parameters in an encrypted form in the database.

```
public void SetPeerPublicKey(byte[] peerPublicKey)
{
        PeerPublicKey = peerPublicKey;
        SharedKey = Curve25519.GetSharedSecret(PrivateKey, peerPublicKey);
        HmacKey = SharedKey;
        DerivedHmacKey = Curve25519.GetSharedSecret(HmacKey, HmacKey);
        using var aesProvider = new AesCryptoServiceProvider
        {
                Key = SharedKey
        };
        aesProvider.GenerateIV();
        IV = aesProvider.IV;
```

```
}
```

The server's public key alongside with the *IV* is then included in the response of the login request. This way the client is able to generate an identical shared secret using the same methodology. In the end of the authentication process both sides will possess a secret encryption key and hash parameters despite not sharing them directly. These secret values are used whenever a document is shared between the two parties. In addition to the encryption of these documents, the application makes use of the hash-based message authentication (*HMAC*) in order to ensure data integrity and authenticity.

Transport Layer Security (TLS)

The third and most general layer of security can be found on the transport layer in the form of *HTTPS*. This layer is responsible for providing communications security over the network using a self-signed *SSL Certificate*. It enables communication bewteen parties in an encrypted form and it is used in all of the application's requests.

Authentication and authorization

The majority of E-me's endpoints require authorization to be accessed. In order to authenticate and authorize a user, the application utilizes *JSON Web Tokens (JWT)*. These tokens are generated by the authentication service upon providing the correct login credentials from the application frontend and are to be included in the header of the HTTP requests by the client. These tokens contain information about the user, the generator of the token and timestamps which are used to verify a token's validity. On the server-side, the *JwtTokenValidatorMiddleware* is responsible for processing the request header and validating the incoming tokens for each request.

4. Chapter

Results and evaluation

Summary: *In this chapter I describe decisions I made, difficulties I faced during development and the quality of my code.*

4.1 Metrics

In this section I will evaluate some of the algorithms used in the application, test coverage, code analysis. 4 pages

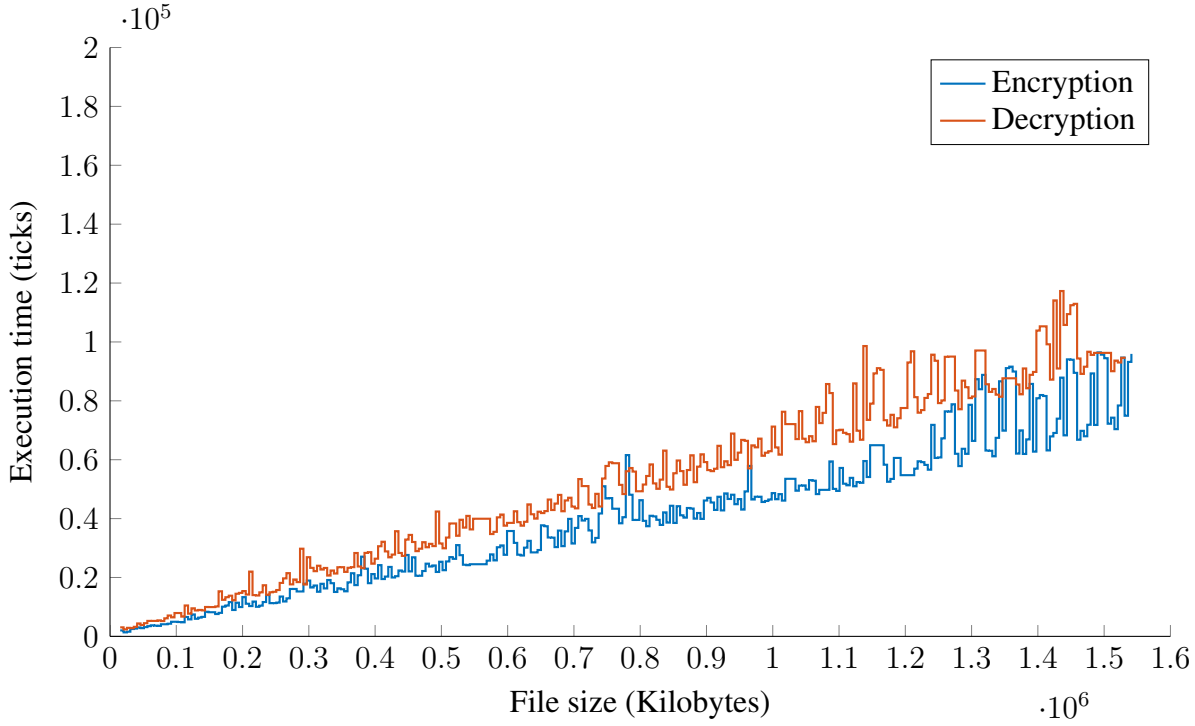

Figure 4.1: Encryption/Decryption duration over document size

- chart about the duration of the encryption (time versus file size)
- service-level test coverage
- code metrics

    * maintainability
    * cyclomatic complexity
    * average depth of inheritance
    * average class coupling
  - system requirements
    * server
    * mobile

4.2 Decisions

Here I describe decisions I made about what technologies to use, what I considered using and how they can be replaced with other ones. 3-4 pages

- backend
  * Java
  * Python
- frontend
  * Kotlin
  * Java
  * React Native
- why I chose C# and .NET instead of native languages
- Data storage
  * MySQL
  * MongoDB
  * Oracle
  * Firebase
- Auth technologies
  * Cookie-based auth
  * Multi-factor auth
  * Biometric auth

4.3 Obstacles and difficulties

In this section I contemplate about different parts of the applications that were problematic to develop. 3 pages

- HTTPS on Android
  - difficulties connecting to the server on HTTPS
  - certificate issues
- Windows CNG not being implemented in Mono
  - switching to the open-source ECDH implementation
- accessing resources within the Android secure storage (icons, config files etc.)

4.4 Possibilities

Here I describe possible features, future upgrades for the app. 1-2 pages

- Adding biometric authentication
  - fingerprint
  - face recognition
- ML based form categorization
  - categorize unknown fields based on user inputs
- Requirement-tree for documents
- digital signatures
- notifications (expired/soon-to-be expired documents)
- Administration application
  - validating data
  - granting permission for document release
  - preparing templates
- IOS release

4.5 Retrospective

In this section I review the development process and describe what would I do differently and why. 1-2 pages