Face Recognition System
Design Document v1.0

Authors: Walid Balegh, Jakob Heyder, Sarpreet Singh Buttar, Henry Pap and Oscar Maris
Semester: VT 2017

Contents

1 Introduction

The FRS (Req. 1.3) design document is intended to be the groundwork of how the software should be built. Members of the development team should be able to read this and have a similar understanding of how the software should be developed. Details of the known unknown will also be documented here.

1.1 Purpose

The design documentation of the FRS is intended to create a mutual understanding of the software's functionality and contents. This document should contain all necessary information to start development of the face recognition system. This includes all known technologies, pitfalls and the architecture that are necessary to the FRS as a whole.

1.2 Definitions, acronyms and abbreviations

- **FRS : Face Recognition System to be developed** (Req. 1.3)
- **PN : Swedish Personal Number** (Req. 1.3)
- **User-Entity: A User entity consists of an identifier(ID), a personal number and a link to a picture of the person** (Req. 1.3)

See Requirements document for related acronyms and references.

1.3 References

Requirements Elicitation v1.0 of the Face Recognition System for LNU. Often mentioned as Requirement Elicitation, Requirements or "Req.", which a referenced chapter.

1.4 Priorities

Security: As we are dealing with users private information it is imperative that the system is both secure and has limited access for both users and administrators. The primary methods to achieve this will be: The use of HTTPS SSL/TLS 2.4 and both admin and user authentication.

Reliability: When the user requests a PN from the server there needs to be a certain guarantee that that the PN returned matches the image provided. In order to accomplish this, the skybiometry api will have to meet a certain percent level of the two compared faces. The required percent level will be parameterized and on request can be defined and changed by the customer.

Speed: The primary function for the user side of the FRS, the GET request (Req. 3.1.5) is constrained to have a five second process time when returning a PN. (Req. 2.4) The relevance of this number is to ensure user friendliness to the client. Since this relies heavily on the response time of the external API, this must be archived in cooperation with skybiometric. This does not effect further design decisions, since the requests for a single entity have to be handled synchronously and the process time is highly dependent on the production-deployment environment. Further handling and validation will be part of the testing phase for production deployment.

1.5 Overview

This design document is for a face recognition software. The FRS purpose is to provide a PN (Req. 1.3) using a comparison of a user provided image of a person's face. There should be authentication to both users and administrators. Administrators should have access to a list of the PNs from the database. Administrators have the functionality of add, edit, list and delete user-entities from the database (Req. **??**). Additionally to the FRS an example user application, known as the UAM (Req. 1.3), will be developed to present the functionality of the systems API. Said application should include the ability to authenticate the user, request PNs using images and show the CRUD functionality of the administrator.

2 Major Design Issues

In this section all major design issues and decisions will be discussed. Rational and alternatives will be presented. The detail of evaluating the alternatives depends on the trade off the decision has.

2.1 Architectural Design

As desired from the customer the system will have a Client-Server structure. Where the in section 1.3 of the Requirements Elicitation defined UAM and AAM are the client modules and the URM and ARM are the server modules. Further we may reference URM or ARM as Server and UAM and AAM as Client. It is important to note that as defined in the requirements Req. Doc. 2.1.1 our developed product is only the Server with a clear defined API and the Client application is the desired example client to show how integration and usage could be done. The architecture of the server is defined in detail in Section 4.

2.2 Languages & Frameworks

In this section Technology's for the server concerning language and frameworks will be reasoned and concluded.

2.2.1 Server-side

As choice of programming language we choose the **Java programming language**. It is one of the most widely used ones on the server side. Especially the in 2.2 of the Requirements Elicitation defined REST functionality is supported by various frameworks with large communities. Rationals which led to the choice are listed below.

- Great flexibility & portability due to platform independence with the JVM.
- Scalable solution based on proven enterprise solutions
- High productivity because of existing frameworks and solutions
- Good support by a large community of developers

The Spring Framework will be used on the server side of the system. More detailed the Spring Web, Security, Data and Boot modules. The Spring Framework allows fast, enterprise scale development of applications by providing features for security, RESTful applications and Data Management and thus is the perfect fit for the requirements defined in the Requirements Elicitation 2.2 and 2.4 such as authentication, encryption and REST functionality. The Rationals for using the different modules are listed more detailed below.

- Great Portability and Integration - it is supported by various cloud providers to make deployment and continuous development possible.
- Spring Boot provides an embedded application server which allows fast and easy setup of an application.
- Configurability - Spring is easily set up and gives good default solutions but also provides the possibility to configure the details to the application needs
- The Spring framework provides RESTful support which is asked for from the customer and fills the application needs.

- Spring Data provides a convenient way to implement CRUD functionality for accessing and modifying data. It supports various Database technologies such as JPA and generates boilerplate code at run time which reduces developing costs.
- Spring Security provides enterprise ready security features for authentication and encryption without much setup.

2.3 External Services

The external services needed to develop the FRS will be discussed and covered here. If there are multiple technologies that could be considered, they will be reasoned and selected within this section.

2.3.1 External Face Recognition API

One of the core features as defined in 2.2 of the Requirements Elicitation is to match a photo of a face with an existing one to a certain factor of equivalence/similarity. For this purpose as defined in the requirements an external service will be used. Following we compare some of the most known ones. Important aspects are cost factor, usability for the specific needs and security support. There are some External Face Recognition API out there such as

- *SkyBiometry:* It is a cloud-based face detection and recognition software which provides a high-precision biometric identification for over 20 years. In addition, it also provide API client libraries in various languages such as Java, C#, Pyton etc for giving a quick start to the developers. Regarding the usage limits, it has a free subscription which allows 100 methods calls hourly and 5000 monthly. It provide SSL support and its API uses REST interface which means all the API methods are called over the Internet using standard HTTP methods and responses are generated in XML or JSON.
- *Lambda Labs:* It permits developers to send an image link to their service for the identification. In addition, it also allow to create an album of photos, analyze and compare new images with existing ones. Regarding the usage limits, it has a no free subscription and minimum cost is $9/month. It does not provide SSL support and its API also uses REST interface and responses are generated in JSON.
- *OpenFace:* It is a open source web service which provide facial detection technologies. Its API uses REST interface and accept image from the developer and return a JSON response. Currently, it does not support SSL and can only detect up to 80 points on a given image.

We have found that *OpenFace* does not provide sufficient functionality as compared to others. Wheres *Lamba Labs* does provide needed functionality but with a cost of $9/month. In result, ***SkyBiometry*** is the free and suitable option for our project.

2.4 Communication Technology

All technologies and protocols regarding communications will be discussed in this section.

2.4.1 Authentication

Authentication as defined in 2.4 of the Requirements Elicitation will be done by providing a username and password for registered Users and Admins. As defined in the Req. **??** the software will be used by institutions and companies with high security issues. Thus the registration will be exclusive over a non automated channel by contacting the Customer/Developers to verify a service. Further automation or implementation can be discussed with the customer in the future. The in 2.4 the Requirements Elicitation mentioned credentials refer further to a user name and password. The authentication will be session based for usability. However the session will only be saved on the client side and the server will be completely stateless as defined in the standard for REST applications.

2.4.2 Data format

The data will be formatted in standard JSON as represented in the official API. (Ref. API document) The format is human-readable and widely supported. It also supports the requirements of a RESTful application.(Req.Elic. 2.2)

2.4.3 Communication

The Communication will be over IP/TCP to have reliable transport and uses HTTPS on the application layer. This ensures general security by using SSL/TLS and ensures data integrity and privacy by authenticating the application. It provides the encryption and security defined in 2.4 of the Requirements Elicitation.

2.4.4 Encryption HTTPS SSL/TLS

The face recognition system will be using HTTPS using the SSL/TLS transport layer protocols. Implementation will be uncomplicated as MySQL supports TLS natively.

HTTPS: The **H**yper**t**ext **T**ransfer **P**rotocol is the current industry standard for communications over the world wide web and is also the main communications method used by the skybiometry api. The images and personal numbers sent over the FRS will have to be encrypted therefore the encryption method will be HTTP**S** (Security) - SSL/TLS, an extension of HTTP.

SSL 3.0: Secure Socket layer 3.0 is the previous security protocol used for HTTPS. SSL is no longer secure enough for commercial use such as the FRS. Therefore the FRS will implement the most recent successor to SSL, TLS. TLS is still often referred to as SSL which is why the differences must be made clear.

TLS 1.2: The Transport Layer Security 1.2 is the current standard used in today's online banking software. The FRS will be implementing TLS 1.2 as it's security protocol, as it is both the most recent of its kind and has been tested and used in online commercial software. It is good to note that TLS 1.2 is no longer supported by certain windows XP and vista servers. For that reason the older version TLS 1.0 could be considered.

Certificate: The FRS concerns sensitive user information, because of this HTTPS is the only viable solution for encryption, as there are currently no other standards that can guarantee the same security and convenience necessary for the FRS. A SSL/TLS certificate will have to be provided by the stakeholders for implementation. Early builds of the FRS system will not implement the encryption specified in the requirements until a SSL/TLS certificate is provided.

Functionality:

1 The client verifies the TLS certificate and sends hello message

2 The client tells the server the potential encryption methods and the server selects one dependant on the implemented encryption table.

3 The server sends the certificate with the public key.

4 Both computers then calculate a code based on the certificate and encryption methods chosen.

5 The server complies with a final encrypted message and the encrypted communications start.

Providers: Potential providers of a TLS certificate should be reasoned with the stakeholders and include companies such as:

- Symantec
- Comodo
- Digicert
- GlobalSign
- Godaddy

Implementation: MySQL supports HTTPS so simple encryption should not be difficult to implement. However there are two possible implementations of TLS, simple and mutual. If possible a Mutual implementation of HTTPS should be considered, as mutual is more secure but requires a personal client certificate in order to work. However the registration is required to be for selected institutions with high security (Req. **??**) and not open to any, thus the implementation of this certificate can be controlled by the development team.

2.4.5 Comunication speeds

The FRS is to be required to have a 5 second response time when retrieving a PN via photo 2.4. This entails the management of both upload speeds and the external SkyBiometry api.

SkyBiometry optimization: As SkyBiometry is an external api and is not developed by the team, it is hard to predict how fast images are able to be compared with a larger database. There are however, a number of methods in place to optimize face recognition speeds. The two primary methods that will be used with skybiometry are DETECT and RECOGNIZE. For any further details then these functions, see skybiometry documentation.

DETECT :The DETECT function of skybiometry will detect and return tags for a given face. To minimize the amount of faces to be processed within the database the FRS system will have to save and sort each photo using the skybiometry tags. Tags to be considered include: geometric information, eyes, nose, mouth, gender and age. When admin calls the

ADD function photos added should be processed using the skybiometry detect function. Images should then be sorted by tag within the database. This way when users call the primary GET function we can use the skybiometry DETECT function to minimize the amount of images that will have to be processed by the skybiometry RECOGNIZE function.

RECOGNIZE: If the sorting of tags is handled correctly there will be fewer images for the FRS to sort through. The RECOGNIZE method has parameters for both normal and fast scan. As the DETECT function will have sorted images by tag it may not be necessary to use the normal parameters and then fast can be used instead. Accuracy will have to be tested with these functions when the skybiometry api has been implemented. Lastly the skybiometry api rescales larger images to match 1024 pixels width and height, so it maybe more effective for images to be smaller or equal in size.

2.5 The example client

2.5.1 Client-side

The example client application as defined in 2.1.2 of the Requirements Elicitation has to be working on mobile or web browsers. It is therefore suitable to make a web browser interface which is also responsive on mobile. Thus the application will be able to run on iOS, Android, Microsoft phone and WEB. Alternatively it could be developed with one of various Cross Platform Mobile development tools out there. Some of them are listed below.

- **Xamarin** - which is the most popular choice, a free trail is available and it use the language C#. This will make it more structured as C# is an *OO* language.
- **Phone Gap** - which is the most well known tool, it is open source meaning that it is free. It uses the common web languages to create hybrid apps i.e. HTML, CSS, JavaScript.
- **appcelerator** - lets developers use JavaScript to build their apps, provide mobile testing, it has a GUI to create design (which uses common HTML and CSS), and lastly it is free.

There is plenty more but since we only want to show the usage of the API by example as written in the Requirement Elicitation 2.1.1 we will not need a native application. Using common WEB languages to create a browser alternative fulfills the requirements. No requirements constrain the decision among various alternatives for the Web development. Thus this will be discussed in the next section 3

3 Technology choices

In this section we will discuss technology choices which are not constrained by the requirements and thus do not belong directly into the design space defined in 2.

3.1 Client technology

In this section we will shortly discuss the technologies used for developing the WEB Client. *It is important to mention that this is not a client in the classical sense as that the browser is the communicating instance and the client only consists of the pages served from a web-server. So it is more only a user interface to access the API.*
We will use responsive design to style our app so it will be desktop, tablet and mobile friendly. For this we will use a CSS framework, all CSS frameworks comes also with a JavaScript framework for design purpose (animation etc.). Further a list of recent CSS frameworks:

- **Bootstrap** - the most common framework out there, easy to use and creates fast design. Great for dynamic designing thanks to its grid system. The major drawback is that it will look boring and old.
- **Material Framework** - Google's own framework, a google look alike framework.
- **Semantic UI** - a fresh framework that has grown quite popular in the last couple of years. It uses the JavaScript jQuery framework which is easy to use and has great AJAX calls which can be helpful.

This is only a design option and we will go with the Semantic UI because it has a suiting design, and for the use of jQuery.

To summarize The framework that will be in used for WEB development is Semantic UI for CSS and jQuery for JavaScript as it comes with Semantic UI.

3.2 Development Platforms

This section will cover the platforms used to develop each major component. Such as, Server, client and databases used. This will also include reasoning and conclusions for each platform chosen.

3.2.1 Platform Server

The Server will run on a cloud platform. This gives several advantages which are listed below. Especially easy setup and management are essential for this project during development and by using Java the components are platform independent which allows later changes during production.

- Easy to manage and setup - no System administrator needed
- Allows fast and continuous development and testing
- Cheap and scalable solution

For the development in the cloud, Heroku is among AWS, Microsoft Azure, Google application engine and others a common choice. It supports good conditions for development and support frameworks for features such as database deployment. This gives a convenient way to get the system fastly up and working. Listed are features it provides.

- Native support for Java and Spring Boot application deployment
- Addon support for rational Databases (e.g. MYSQL)
- Github integration for continuous development
- Free use for small scale applications (development)

3.2.2 Platform Client

The Client, more specific the in the to be developed UAM will be Web compatible. Since it is written in Javascript for mobile development it runs on iOs, Android and every system supporting modern web browsers.

3.3 Database

The database will be MYSQL a rational database. It is one of the most used rational databases and therefore provides sufficient features, support and scalability for the application. It is also compatible with the used Spring framework and the Heroku cloud platform. It validates data and ensures integrity.

4 Architecture (Component Diagram)

The subsystems of our system must be reusable and independent from each other, as that will make it easier to implement and test. The different subsystems or components are then combined by the application (which is our system). We define the different components in the component list, then we explain each of them briefly.

Component list

- authenticate
- admin
- user
- face
- database
- storage
- skyBiometry
- imgur
- MySQL

Component details Now we will discuss why we need the different components, how they will be used and what they will require.

1. authentication The authentication component is required for security and divides the system based on whether the connected client is an user or an admin (Req. **??**). The component will take the credentials - username and password - and validate if they belong to an account with user or admin role to access the requested resource. To validate the credentials a query to the database have to be made, thus the required interface is the database component. The server is stateless as defined in the standard for RESTful applications, thus the client is fully keeping track of a local session and authenticates every request.

2. admin The admin component is a core component that will hold all the admin responsibilities which is management of user-entities. The required interface is the face, storage and database component. The provided interface is the CRUD methods as defined in Req. **??**.

- CREATE : create a new user
- READ : get the user information by id or all.
- UPDATE : updates an existing user by id
- DELETE : removes an existing user by id

4. user The user component is also a core component that will be responsible of getting a PN based on a image (Req. **??**) in the requirements elicitation. The required interface is the face, database and the provided interface is "*uploading a picture*".

4. face The face is the main component of this system (FRS), this needs to be as independent as possible in order of our system to be adaptive to changes (i.e. face-recognition algorithm may be replaced in the future for a better one). The required interface that is needed is skyBiometry (the face-recognition api) and the provided interface is CRUD methods (Req. **??**), which allows admins to manage client faces as well as letting user component get a PN based on a picture, see req. **??**.

5. database The database or actually database-access component is the component that will directly talk with the remote database. The required interface is a MYSQL-database since it is flexible and easy to use, but can easily be replaced. It provides CRUD functionality as well as a login method. This is a nessesary component, as defined in 2.1.1 in the requirements elicitation: "*Personal number database*" will also hold information of login-users.

6. storage The storage component is the component that will save the client-face picture to the storage, see 2.1.1, the "*Image Database*" now we will use imgur.com's API to store pictures as space is an issue but in future this can be changed into storing images directly in our server. The required interface is imgur.com-api and the provided interface is upload picture.

7. skyBiometry The face-recognition algorithm api, we will use skyBiometry as discussed earlier.

8. imgur The imgur or imgur.com-api will be used by our storage component.

9. MySQL MySQL is the database type we will use by the database-access component.

Architecture Patterns
We already mentioned that the system will have a client-server structure in 2.1, but now we focus only on the server part as it is the product and client implementations may differ to the given example one. We want good reusability, good cohesion, low coupling, abstraction, security and portability all this because components should be easily changeable, independent so development can be as well and portable so the system can easily be changed to other servers (in the future). Our system is a layer type where the application (which will act as the glue code) will use authentication, admin and user component and these will use the others. See the component diagram to clearly see why it will be a layered system. The patterns that meet our required types i.e. cohesion, coupling, abstraction and portability is:

- Multi-layers
- Service-oriented

The Service-oriented patterns focus is on the communication between the different web-services, in our case is with MySQL database, imgur.com api and skyBiometry. This will use standard communication HTTP and JSON to fulfill open standards. This is a very good pattern to follow when it comes to our remote components.

The Multi-layers patterns focus is on how the communication should be, i.e. layer can only speak with the layer below it, in our case the different components can only speak with the required components and not the other way around (see component diagram to better understand). This is also a very good pattern for our system's structure and supports extensibility and interchangeability.

Conclusion We will use a combined pattern, Multi-layers pattern (for developed component) and Service-oriented pattern (for communication with remote components).

4.0.1 Diagrams

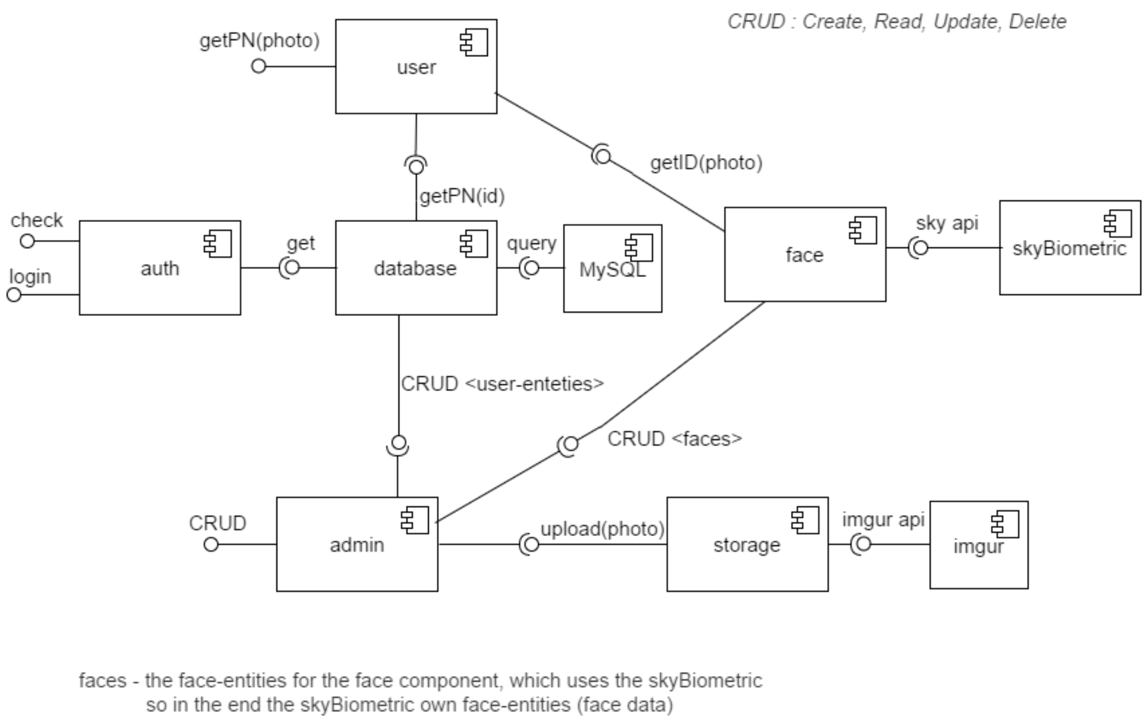

CRUD : Create, Read, Update, Delete
getPN(photo)
user
getID(photo)
getPN(id)
check
login
auth
get
database
query
MySQL
face
sky api
skyBiometric
CRUD <user-enteties>
CRUD <faces>
CRUD
admin
upload(photo)
storage
imgur api
imgur
faces - the face-entities for the face component, which uses the skyBiometric
so in the end the skyBiometric own face-entities (face data)

Figure 1: ComponentDiagram

Component Diagram

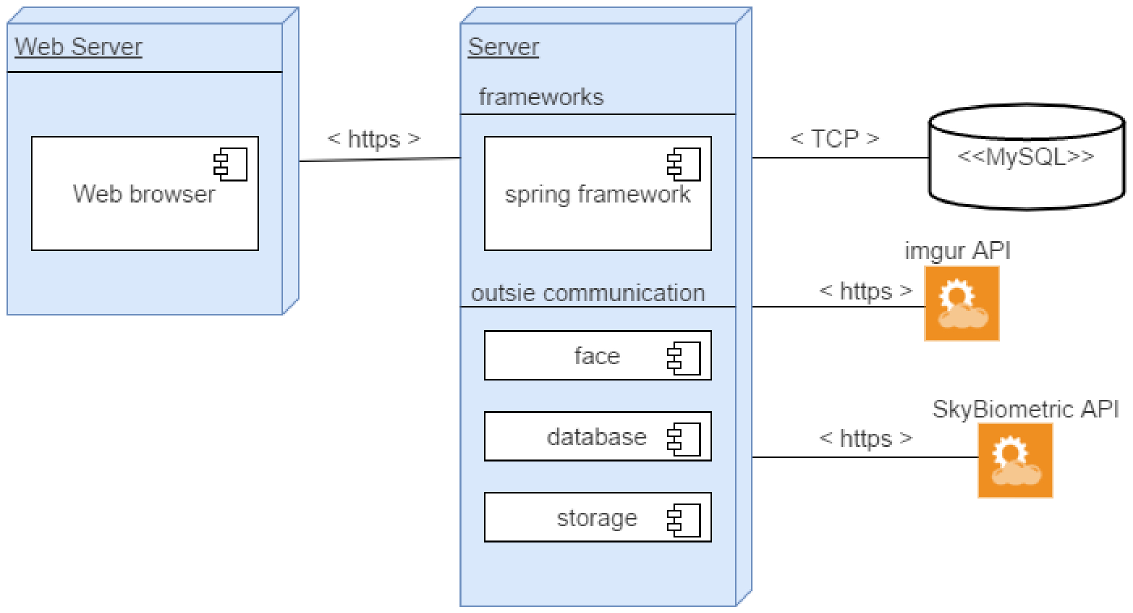


Figure 2: DeploymentDiagram

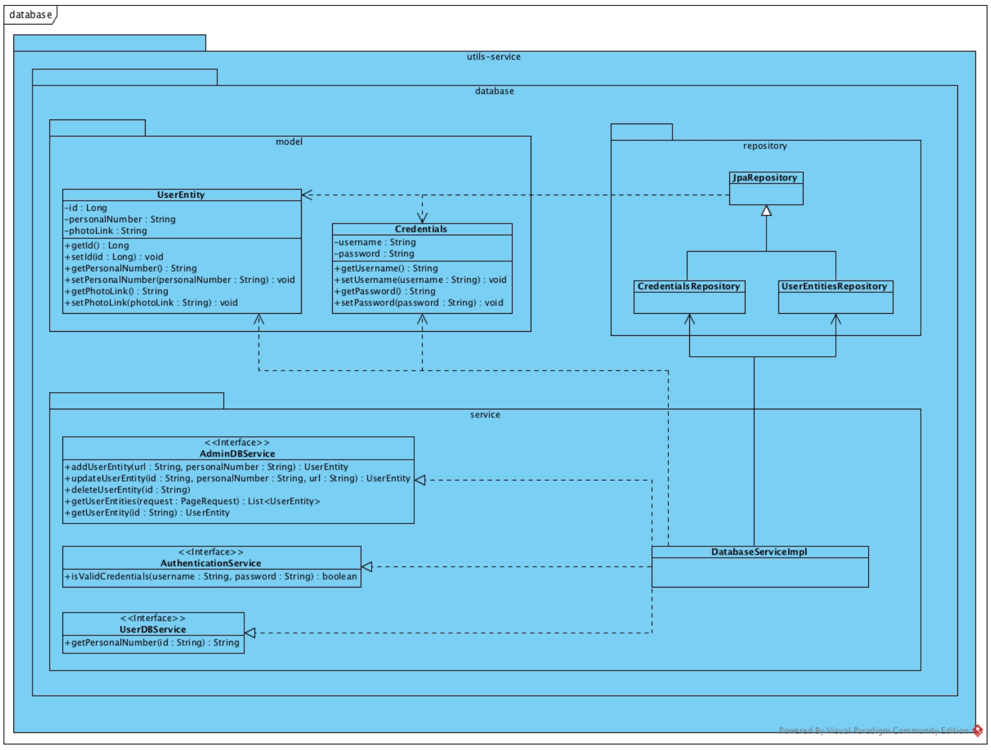
Deployment Diagram

5 Components - Design patterns and static modeling (Class Diagrams)

This section describes the design patterns of each component(Database, Storage, Face, Admin, User and Authentication) of the system. In general, the system will be divided into Model-View-Controller(MVC) pattern where database, storage and face components represents the model; admin, user and authentication components represents the controller and the user who will use the API will be the view of the system (possibly inc. local view-controller). For the purpose of implementation, database, storage and face components will be embedded into one component called utils-service(these components will be independent of each other) whereas user and admin will be implemented separately; and authentication will be embedded into the main component called faceRecognition-service-api where all the components of the system will be integrated.

Below sections will further describe the design patterns of each component as well as its class diagram:

1. Database: It is one of the most important components of the system because it contains the domain of the system and is shared by all the controllers(user, admin and authentication). The biggest problem in this component is to encapsulate the functionality so that user and authentication component cannot use admin functionality and vice versa. To overcome this problem, Facade pattern will be used. This pattern will help in various ways such as database component will provide separate interfaces for each of the controller in order to get encapsulation; controllers will not get any extra information then needed; reduce the complexity of the component etc. Below diagram shows how database component provide three separate interfaces for the admin, user and authentication component.



database
utils-service
database
model
UserEntity
-id : Long
-personalNumber : String
-photoLink : String
+getId() : Long
+setId(id : Long) : void
+getPersonalNumber() : String
+setPersonalNumber(personalNumber : String) : void
+getPhotoLink() : String
+setPhotoLink(photoLink : String) : void
Credentials
-username : String
-password : String
+getUsername() : String
+setUsername(username : String) : void
+getPassword() : String
+setPassword(password : String) : void
repository
JpaRepository
CredentialsRepository
UserEntitiesRepository
service
<<Interface>>
AdminDBService
+addUserEntity(url : String, personalNumber : String) : UserEntity
+updateUserEntity(id : String, personalNumber : String, url : String) : UserEntity
+deleteUserEntity(id : String)
+getUserEntities(request : PageRequest) : List<UserEntity>
+getUserEntity(id : String) : UserEntity
<<Interface>>
AuthenticationService
+isValidCredentials(username : String, password : String) : boolean
<<Interface>>
UserDBService
+getPersonalNumber(id : String) : String
DatabaseServiceImpl
Powered By Visual Paradigm Community Edition

Figure 3: Class diagram of database component

2. Storage: This component is not as complex as database because only admin component will interact with it. This component will interact with one outer component called 'imgur' for saving the images into it. Like database component, Facade pattern is used to provide a simple and clear functionality to the admin component because the admin component does not need to know how this component works. Below diagram shows how storage component provide nice and clear interfaces for the admin component.

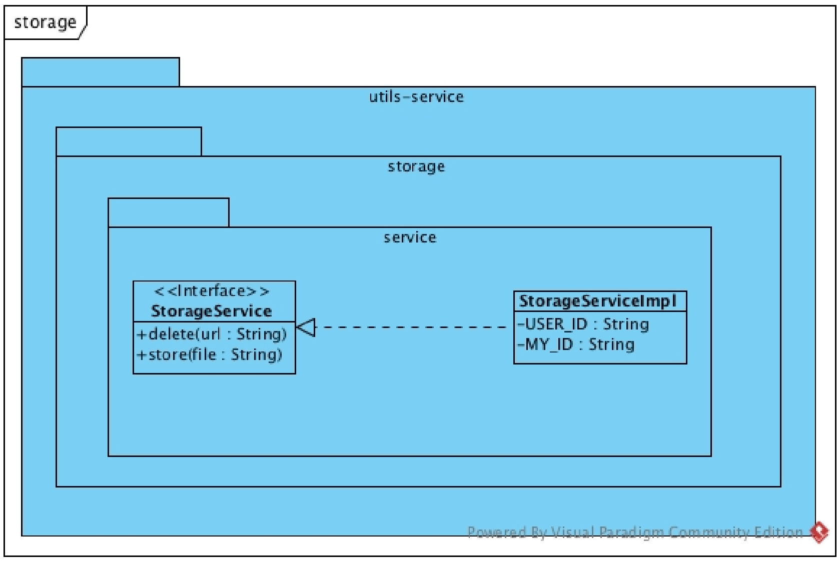


Figure 4: Class diagram of storage component

3. Face: This component is more complex than storage component due to encapsulation problem as seen before in the database component. In addition, it also interacts with the outer comoponent called 'SkyBiometry' for recognizing the faces. Like the database component, it also uses the facade pattern and below diagram shows how this component encapsulate the functionality by providing the seprate interfaces to admin and user component .

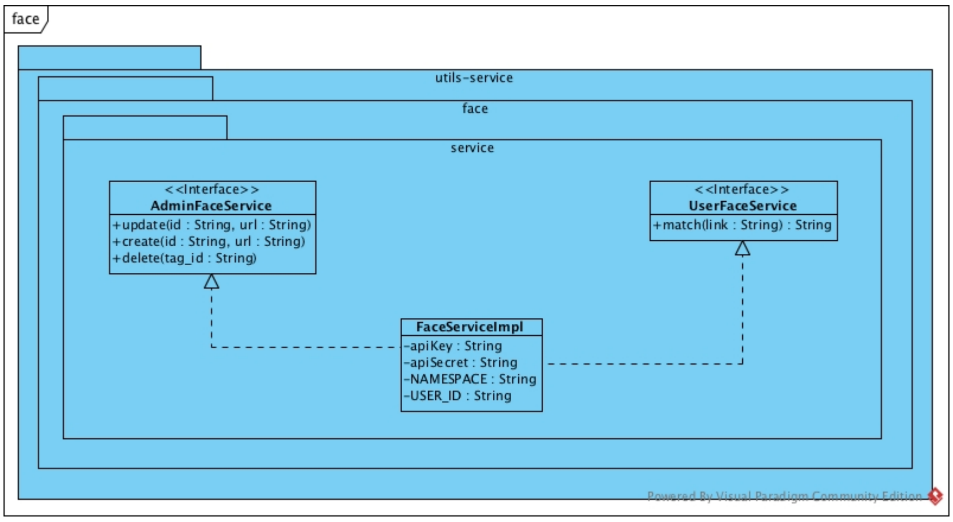


Figure 5: Class diagram of face component

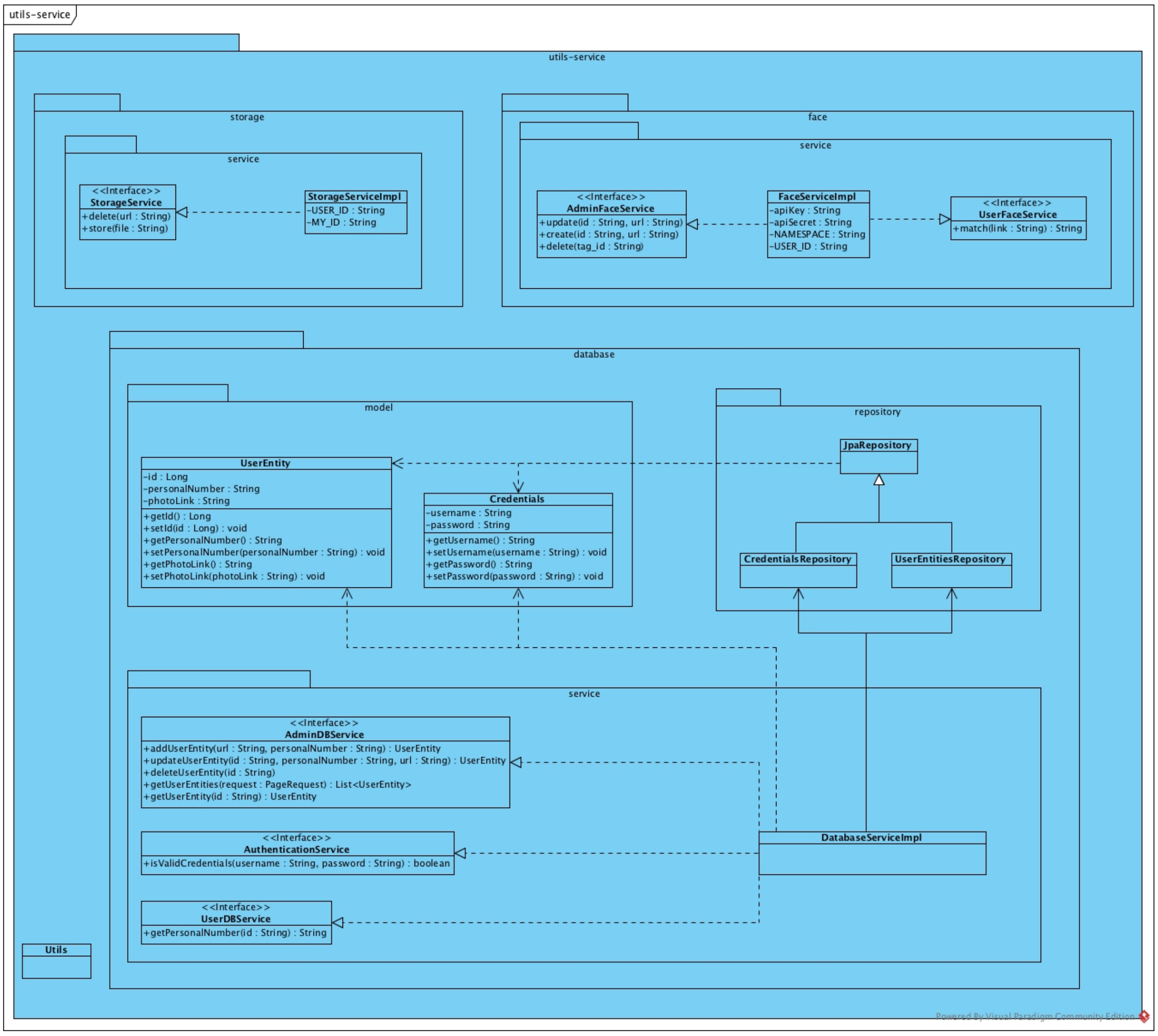


Figure 6: Overview of database, storage and face components embeeded inside the utils-service.
NOTE: ’Utils’ class is for integration purpose and ’JpaRepository’ is a part of spring framework.

4. Admin: This component use most of the important functionality of the system. However, it is very simple because it only structures the order of calls to the interfaces provided by the database, face and storage component. The only problem is to encapsulate the data and implementation from the view because it directly interact with the user API requests. Similarly, like other components, this component use same approach (Facade pattern) to encapsulate the data. It provides an interface to which the view will interact. This pattern helps to make the design more simple, flexible and reliable. Below diagram shows how this component encapsulate the data by providing interface to the controller which will interact with the view(user).

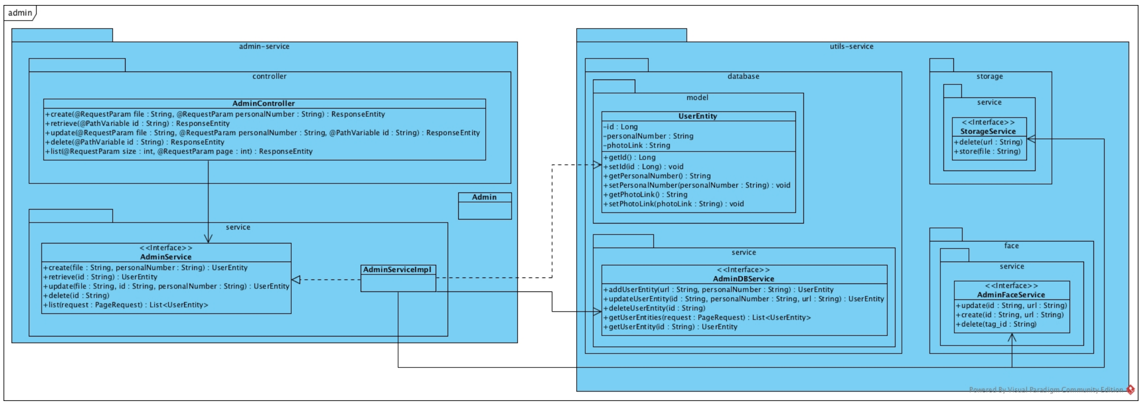

admin
admin-service
controller
AdminController
+create(@RequestParam file : String, @RequestParam personalNumber : String) : ResponseEntity
+retrieve(@PathVariable id : String) : ResponseEntity
+update(@RequestParam file : String, @RequestParam personalNumber : String, @PathVariable id : String) : ResponseEntity
+delete(@PathVariable id : String) : ResponseEntity
+list(@RequestParam size : int, @RequestParam page : int) : ResponseEntity
Admin
service
<<Interface>>
AdminService
+create(file : String, personalNumber : String) : UserEntity
+retrieve(id : String) : UserEntity
+update(file : String, id : String, personalNumber : String) : UserEntity
+delete(id : String)
+list(request : PageRequest) : List<UserEntity>
AdminServiceImpl
utils-service
database
model
UserEntity
-id : Long
-personalNumber : String
-photoLink : String
+getId() : Long
+setId(id : Long) : void
+getPersonalNumber() : String
+setPersonalNumber(personalNumber : String) : void
+getPhotoLink() : String
+setPhotoLink(photoLink : String) : void
service
<<Interface>>
AdminDBService
+addUserEntity(url : String, personalNumber : String) : UserEntity
+updateUserEntity(id : String, personalNumber : String, url : String) : UserEntity
+deleteUserEntity(id : String)
+getUserEntities(request : PageRequest) : List<UserEntity>
+getUserEntity(id : String) : UserEntity
storage
service
<<Interface>>
StorageService
+delete(url : String)
+store(file : String)
face
service
<<Interface>>
AdminFaceService
+update(id : String, url : String)
+create(id : String, url : String)
+delete(tag_id : String)
Powered By Visual Paradigm Community Edition

Figure 7: Class diagram of admin component

5. User: The structure and behaviour of this component is similar to admin component. Again, the only problem is to encapsulate the data and implementation from the view and it uses exactly the same approach as the admin component in order to overcome the problem. The diagram shows how this component encapsulates the implementation from the view by providing an interface to the controller.

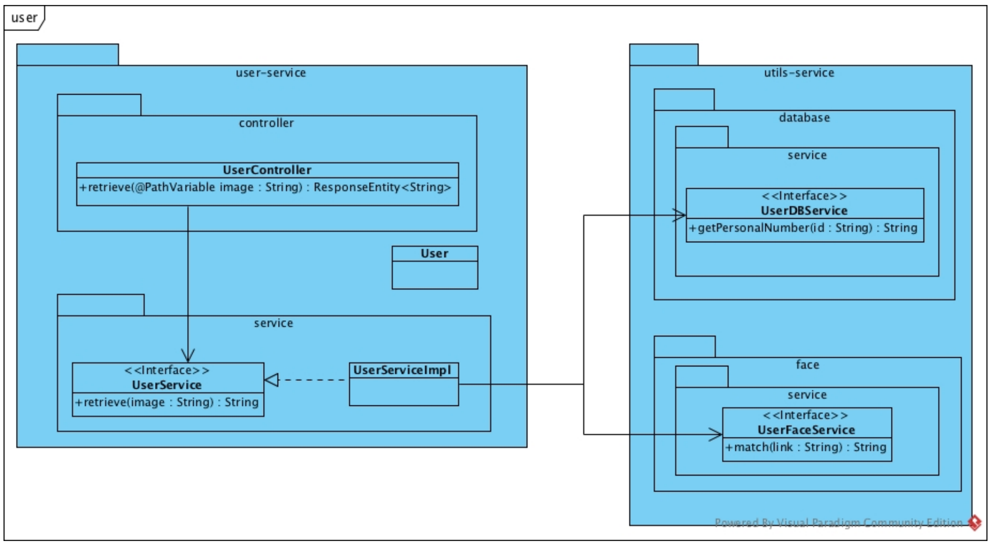

user
user-service
controller
UserController
+retrieve(@PathVariable image : String) : ResponseEntity<String>
User
service
<<Interface>>
UserService
+retrieve(image : String) : String
UserServiceImpl
utils-service
database
service
<<Interface>>
UserDBService
+getPersonalNumber(id : String) : String
face
service
<<Interface>>
UserFaceService
+match(link : String) : String
Powered By Visual Paradigm Community Edition

Figure 8: Class diagram of user component

NOTE: ’Admin’ and ’User’ classes are for integration purpose.

6. Authentication: This component will be embedded into the main component where all the other components will be integrated. This component will mostly use in built security functionality provided by spring framework for authenticating the user. However, for validating dynamic credentials(username and password), this component will use an interface provided by the database component. Below diagrams shows this component structure.

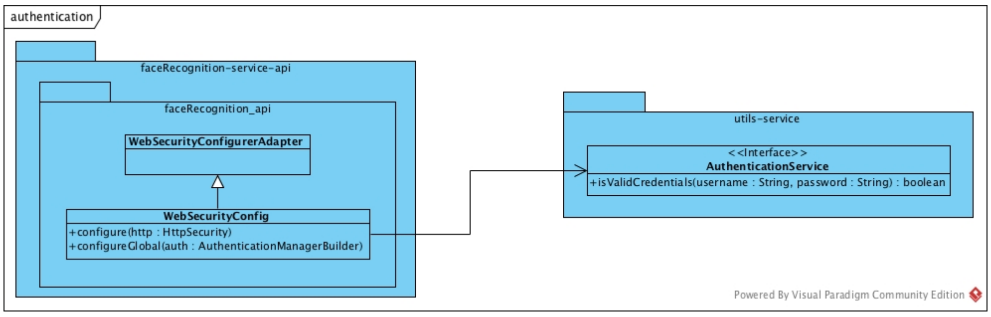

authentication
faceRecognition-service-api
faceRecognition_api
WebSecurityConfigurerAdapter
WebSecurityConfig
+configure(http : HttpSecurity)
+configureGlobal(auth : AuthenticationManagerBuilder)
utils-service
<<Interface>>
AuthenticationService
+isValidCredentials(username : String, password : String) : boolean
Powered By Visual Paradigm Community Edition

Figure 9: Class diagram of authentication component

NOTE: 'WebSecurityConfigurerAdapter' and 'WebSecurityConfig' classes are part of spring data security framework.

Integration: For integrating all the components, we will use inbuild functionality of the spring framework. The diagram below shows the integration structure.

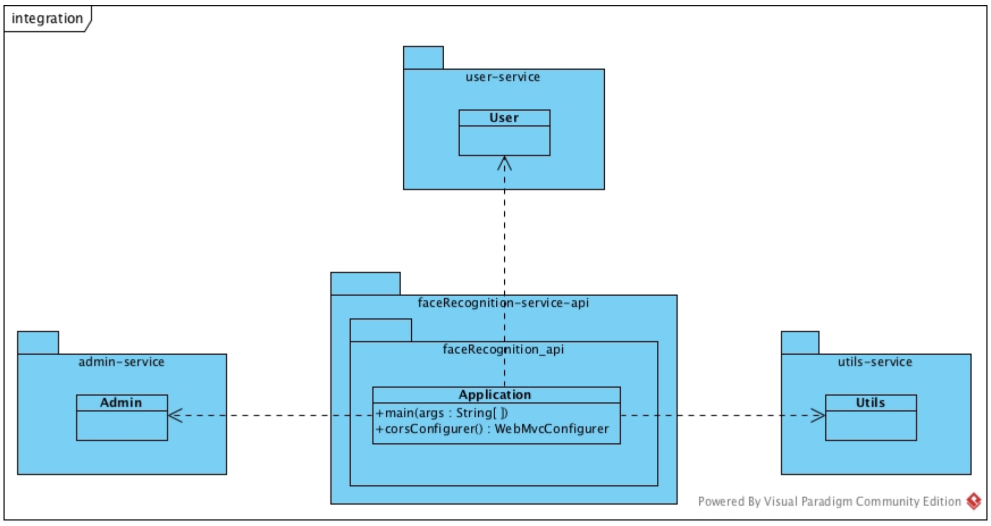


Figure 10: Class diagram of system integration

Complete System: Below class diagram shows the complete system

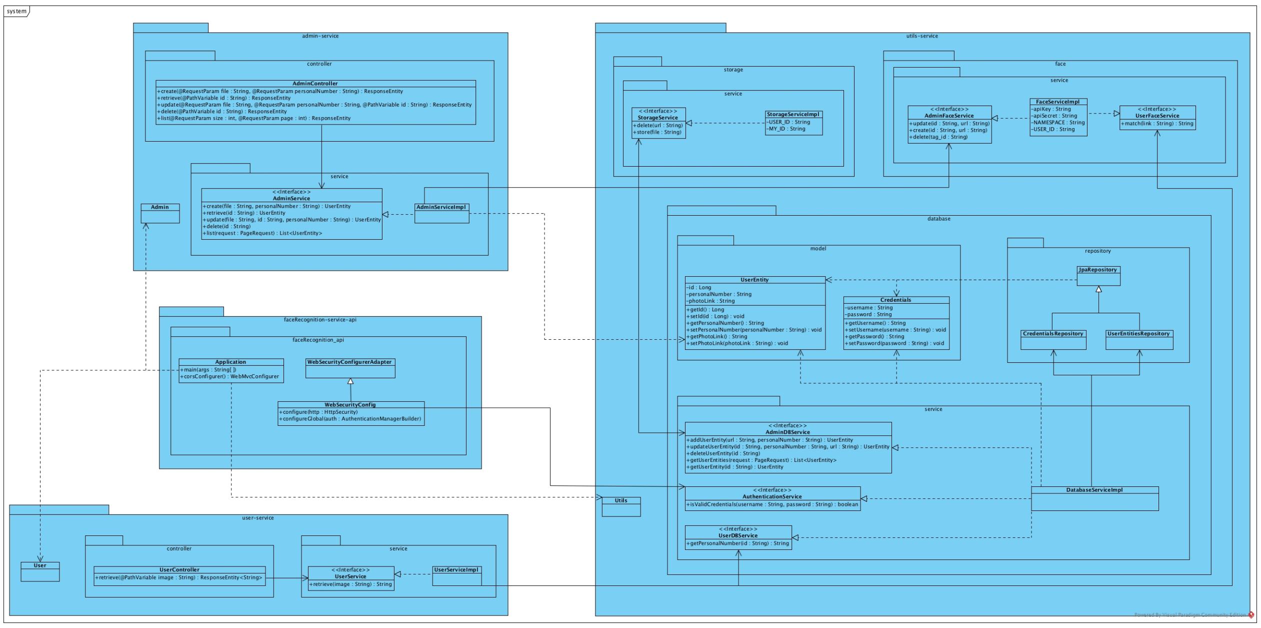
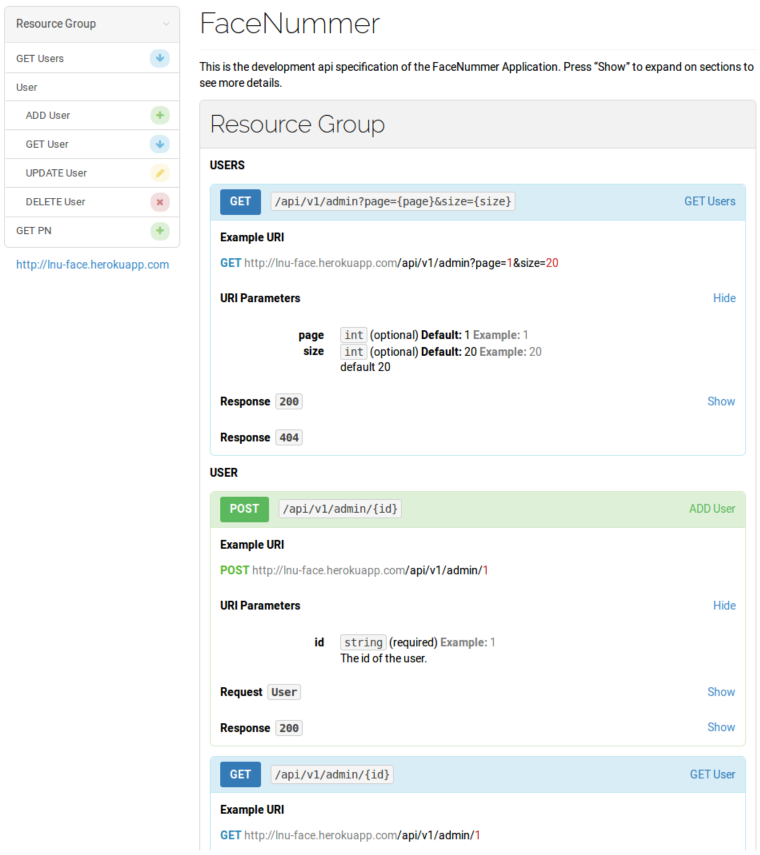

system
admin-service
controller
AdminController
+create(@RequestParam file : String, @RequestParam personalNumber : String) : ResponseEntity
+retrieve(@PathVariable id : String) : ResponseEntity
+update(@RequestParam file : String, @RequestParam personalNumber : String, @PathVariable id : String) : ResponseEntity
+delete(@PathVariable id : String) : ResponseEntity
+list(@RequestParam size : int, @RequestParam page : int) : ResponseEntity
service
<<Interface>>
AdminService
+create(file : String, personalNumber : String) : UserEntity
+retrieve(id : String) : UserEntity
+update(file : String, id : String, personalNumber : String) : UserEntity
+delete(id : String)
+list(request : PageRequest) : List<UserEntity>
AdminServiceImpl
Admin
faceRecognition-service-api
faceRecognition_api
Application
+main(args : String[ ])
+corsConfigurer() : WebMvcConfigurer
WebSecurityConfigurerAdapter
WebSecurityConfig
+configure(http : HttpSecurity)
+configureGlobal(auth : AuthenticationManagerBuilder)
user-service
User
controller
UserController
+retrieve(@PathVariable image : String) : ResponseEntity<String>
service
<<Interface>>
UserService
+retrieve(image : String) : String
UserServiceImpl
utils-service
storage
service
<<Interface>>
StorageService
+delete(url : String)
+store(file : String)
StorageServiceImpl
-USER_ID : String
-MY_ID : String
face
service
<<Interface>>
AdminFaceService
+update(id : String, url : String)
+create(id : String, url : String)
+delete(tag_id : String)
FaceServiceImpl
-apiKey : String
-apiSecret : String
-NAMESPACE : String
-USER_ID : String
<<Interface>>
UserFaceService
+match(link : String) : String
database
model
UserEntity
-id : Long
-personalNumber : String
-photoLink : String
+getId() : Long
+setId(id : Long) : void
+getPersonalNumber() : String
+setPersonalNumber(personalNumber : String) : void
+getPhotoLink() : String
+setPhotoLink(photoLink : String) : void
Credentials
-username : String
-password : String
+getUsername() : String
+setUsername(username : String) : void
+getPassword() : String
+setPassword(password : String) : void
repository
JpaRepository
CredentialsRepository
UserEntitiesRepository
service
<<Interface>>
AdminDBService
+addUserEntity(url : String, personalNumber : String) : UserEntity
+updateUserEntity(id : String, personalNumber : String, url : String) : UserEntity
+deleteUserEntity(id : String)
+getUserEntities(request : PageRequest) : List<UserEntity>
+getUserEntity(id : String) : UserEntity
<<Interface>>
AuthenticationService
+isValidCredentials(username : String, password : String) : boolean
<<Interface>>
UserDBService
+getPersonalNumber(id : String) : String
DatabaseServiceImpl
Utils
Powered By Visual Paradigm Community Edition

Figure 11: Class diagram of the complete system.

API The provided API is summarized inc. data format as an apib and HTML file. This documentation provides all information for a user to build a client application for it. Additionally this can be used forintegration and API testing.

Following shows a snapshot of the API.
The full format can be found as compiled html or apib in API/API_v1.0 .



Resource Group
GET Users
User
ADD User
GET User
UPDATE User
DELETE User
GET PN
http://lnu-face.herokuapp.com

FaceNummer

This is the development api specification of the FaceNummer Application. Press "Show" to expand on sections to see more details.

Resource Group

USERS

GET `/api/v1/admin?page={page}&size={size}` GET Users

Example URI

GET http://lnu-face.herokuapp.com/api/v1/admin?page=1&size=20

URI Parameters Hide

page `int` (optional) **Default:** 1 Example: 1
size `int` (optional) **Default:** 20 Example: 20
default 20

Response `200` Show

Response `404`

USER

POST `/api/v1/admin/{id}` ADD User

Example URI

POST http://lnu-face.herokuapp.com/api/v1/admin/1

URI Parameters Hide

id `string` (required) Example: 1
The id of the user.

Request `User` Show

Response `200` Show

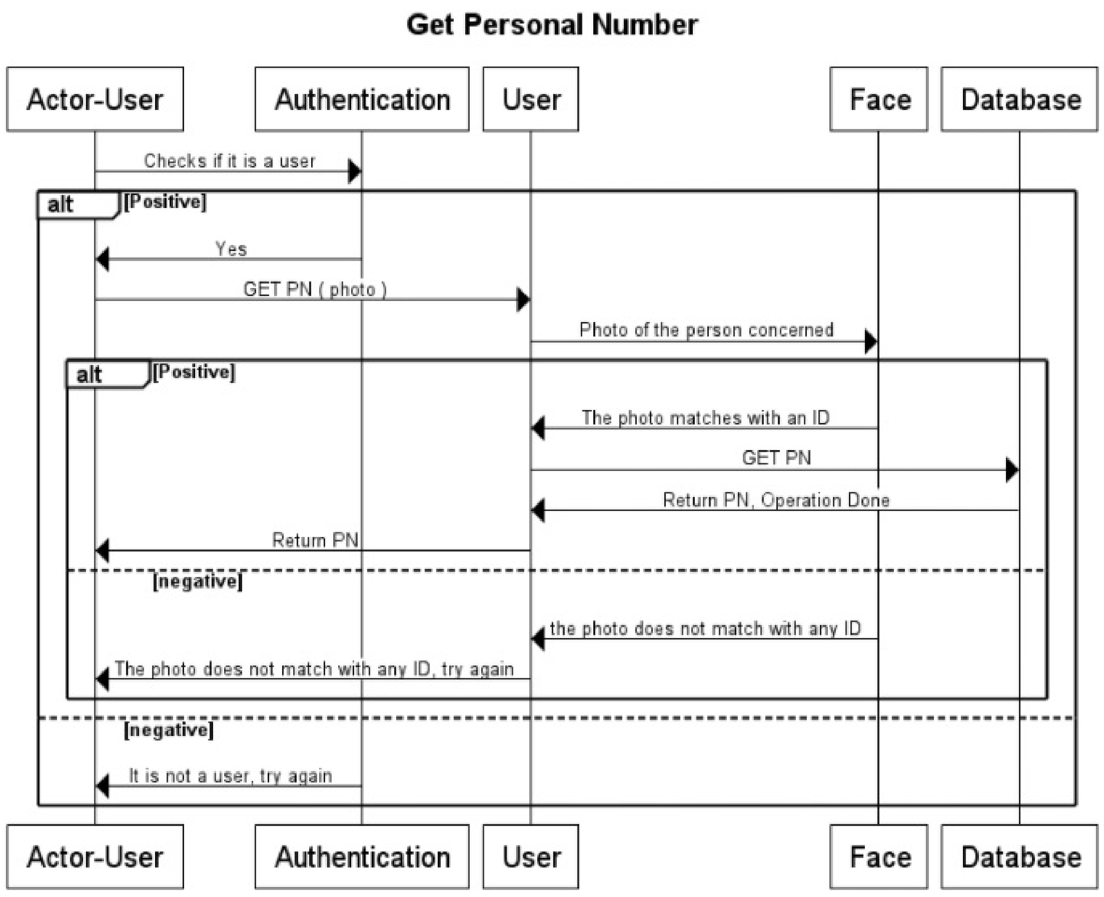
GET `/api/v1/admin/{id}` GET User

Example URI

GET http://lnu-face.herokuapp.com/api/v1/admin/1

Figure 12: Snapshot of the provided API (apib format)

6 Use cases - Behavioral modeling (Sequence Diagrams)



Get Personal Number
Actor-User
Authentication
User
Face
Database
Checks if it is a user
alt
[Positive]
Yes
GET PN (photo)
Photo of the person concerned
alt
[Positive]
The photo matches with an ID
GET PN
Return PN, Operation Done
Return PN
[negative]
the photo does not match with any ID
The photo does not match with any ID, try again
[negative]
It is not a user, try again

Figure 13: Sequence diagram referred to "5.2 UAM: GET Personal Number" in the requirement document.

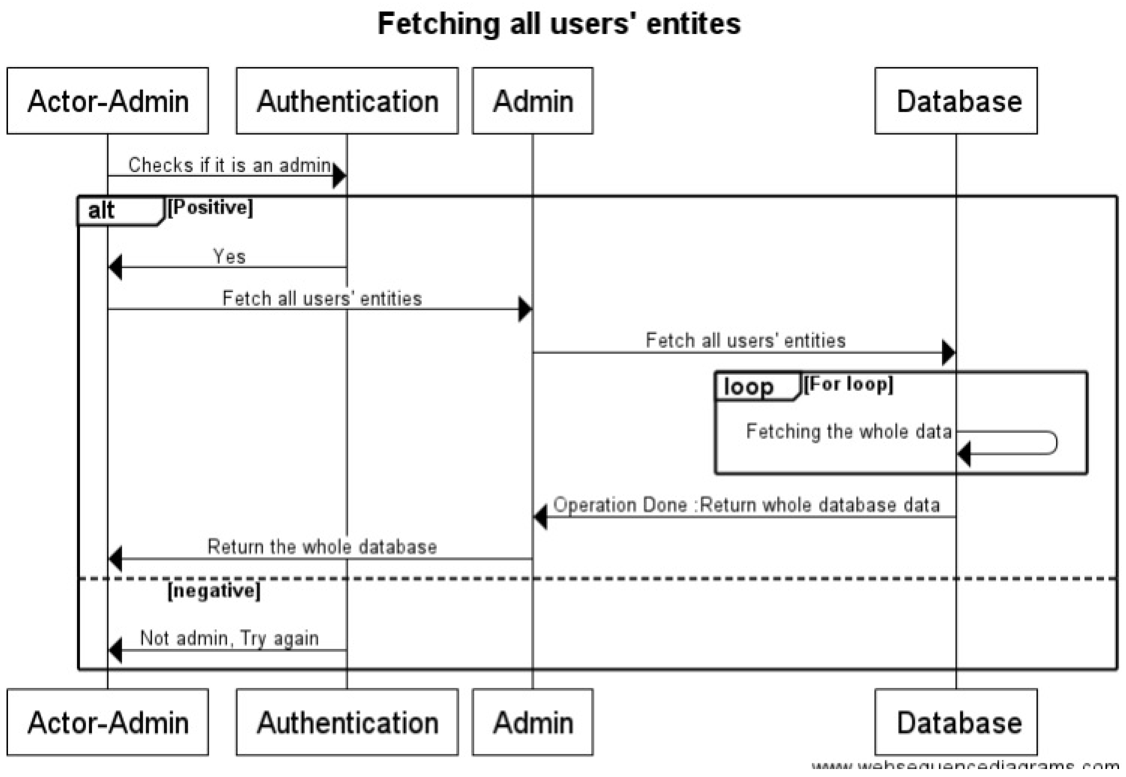


Figure 14: Sequence diagram referred to ”5.3 ARM: Get List of User-Entities” in the requirement document.

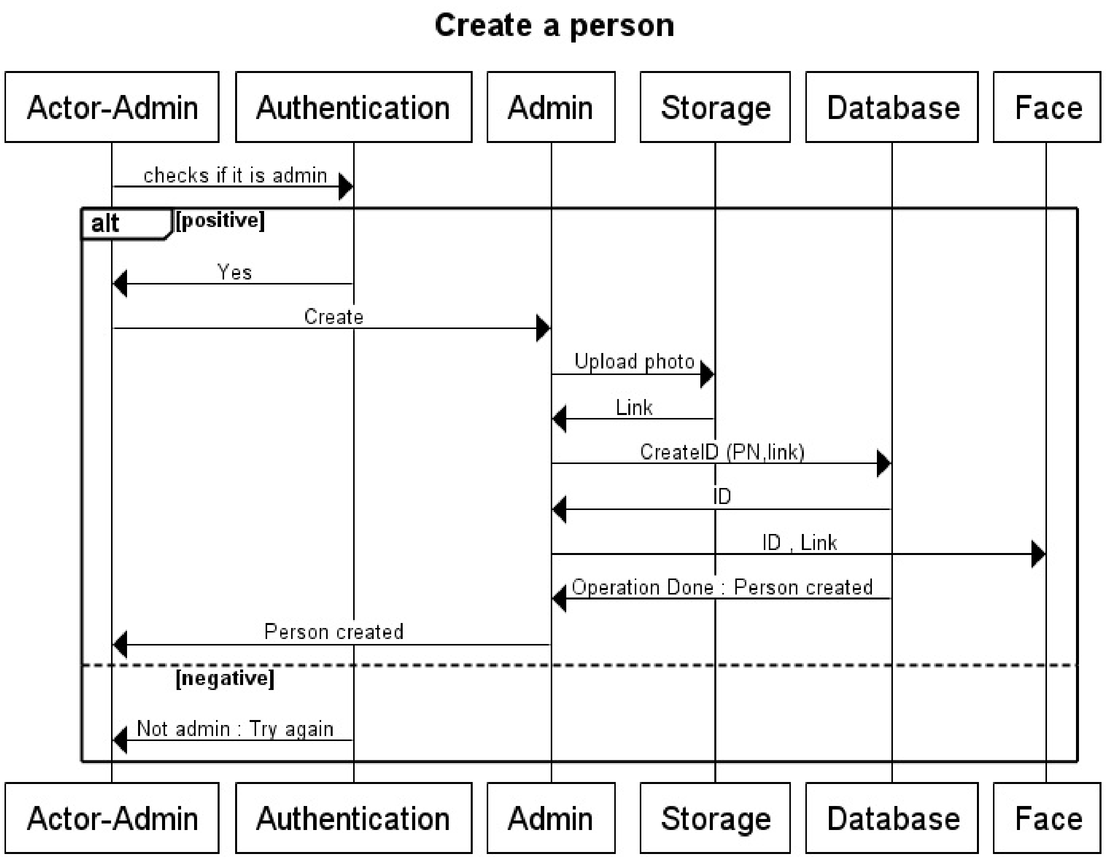


Figure 15: Sequence diagram referred to ”5.4 ARM: Add an User-Entity” in the requirement document.

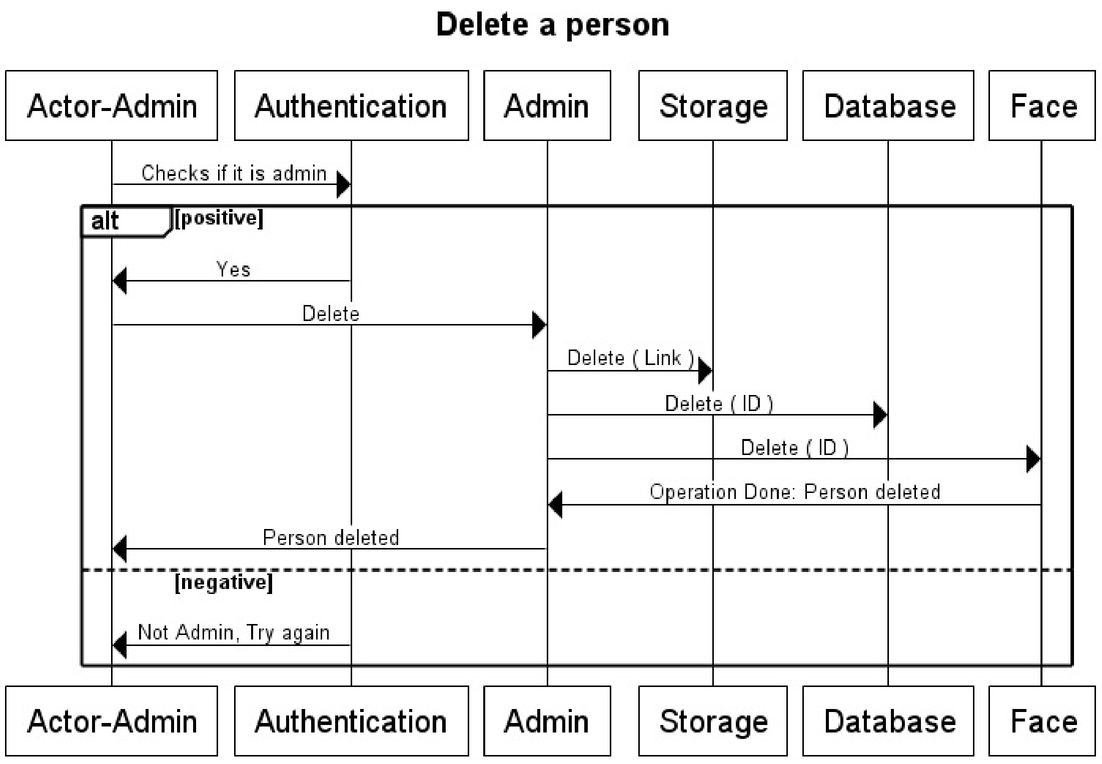


Figure 16: Sequence diagram referred to ”5.5 ARM: Delete an User-Entity” in the requirement document.

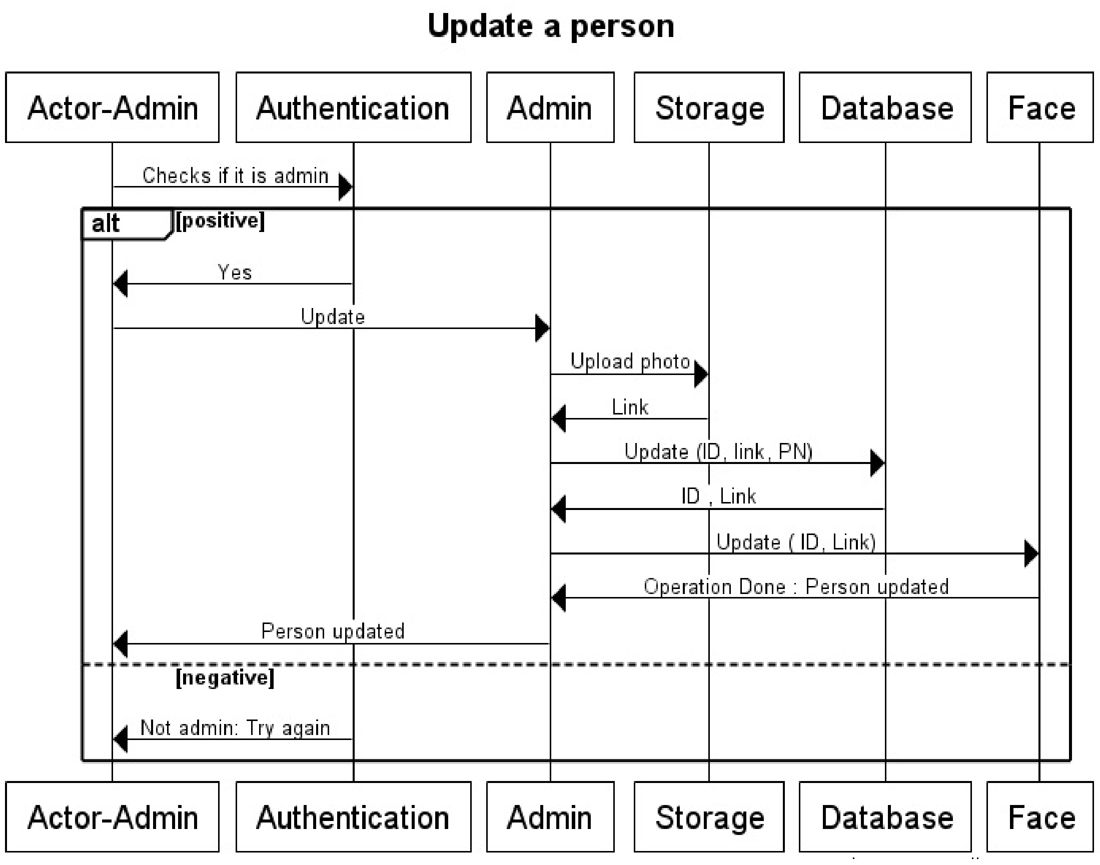


Figure 17: Sequence diagram referred to "5.6 ARM: Update User-Entity" in the requirement document.

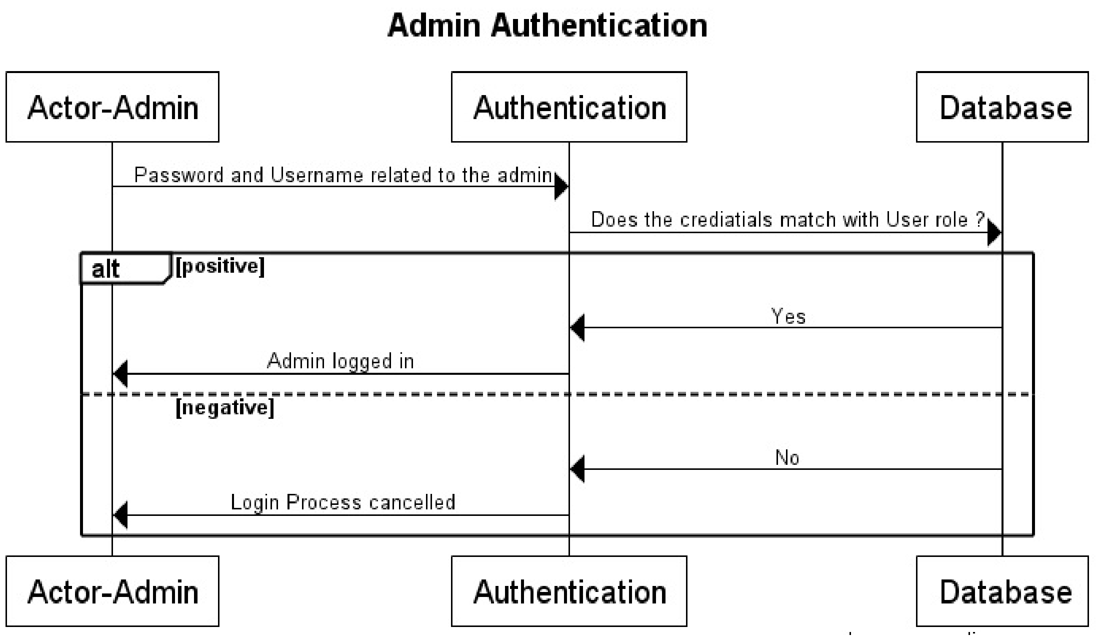


Figure 18: Sequence diagram referred to "5.7 ARM: Authenticate Admin" in the requirement document.

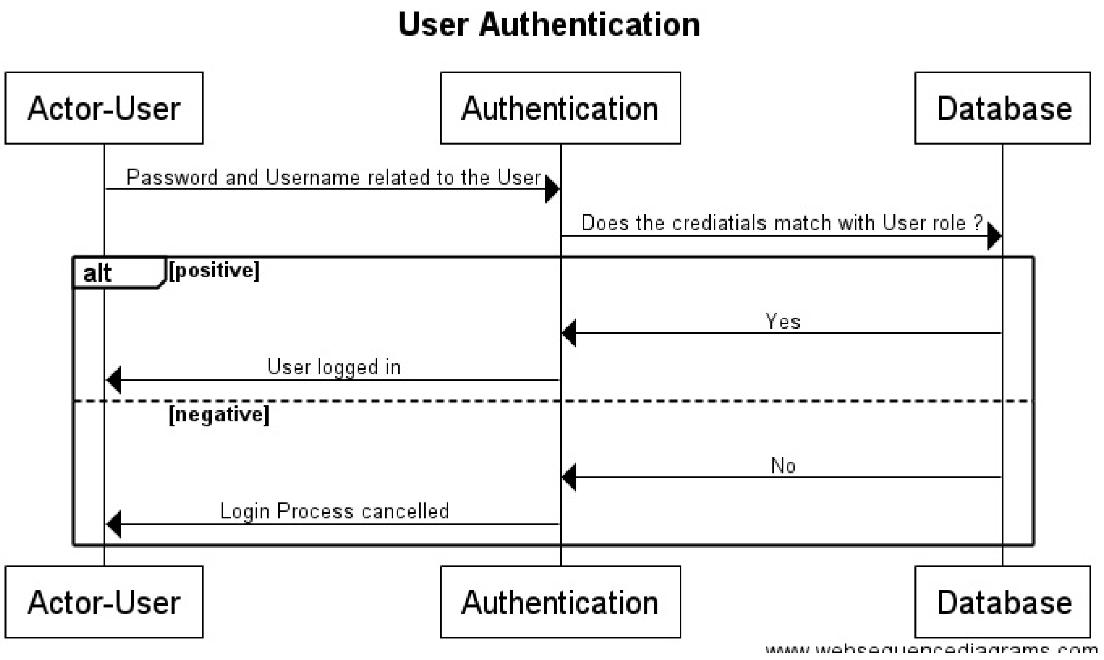


Figure 19: Sequence diagram referred to ”5.8 URM: Authenticate User” in the requirement document.