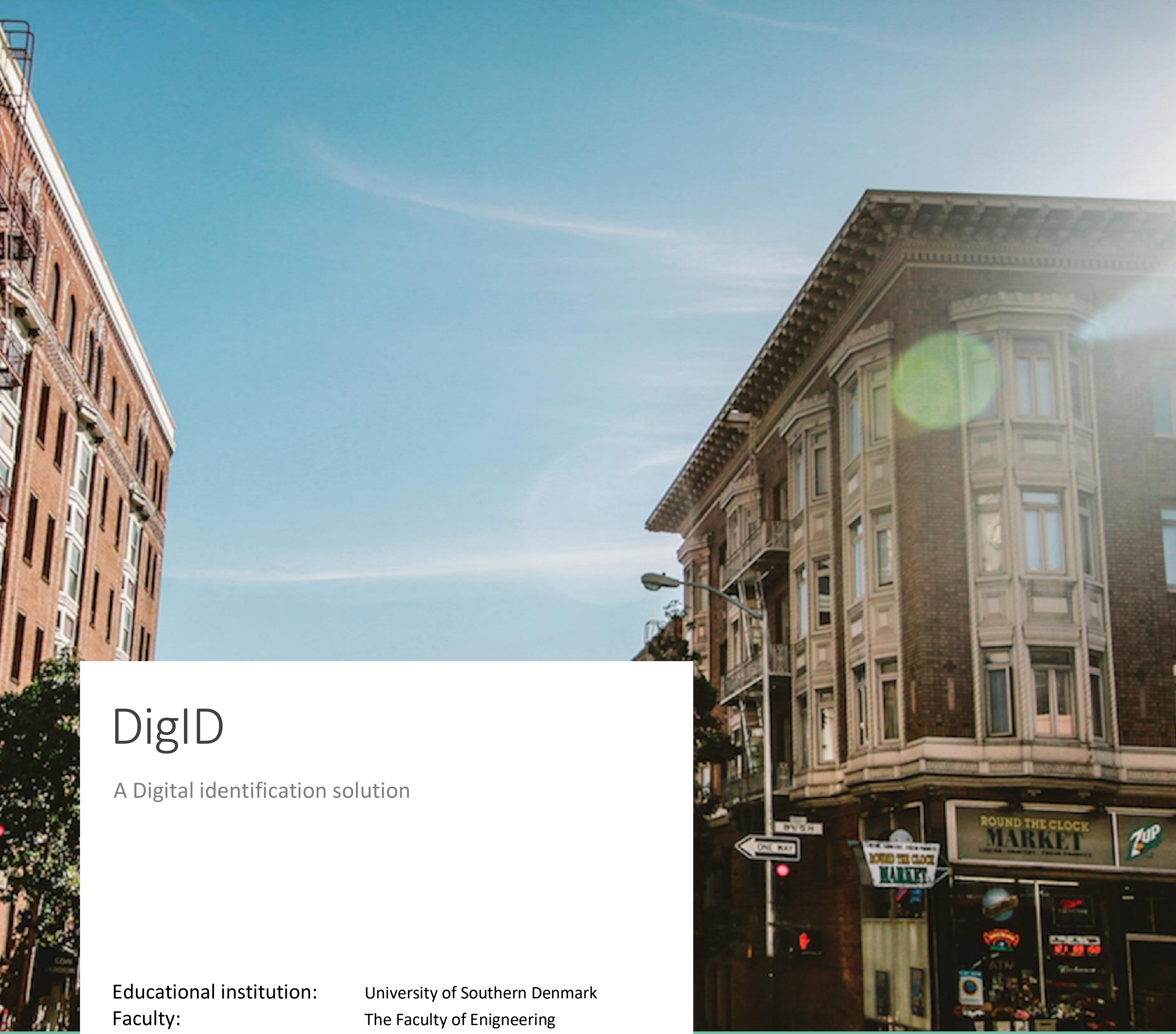


DigID

A Digital identification solution

Educational institution:	University of Southern Denmark
Faculty:	The Faculty of Enigneering
Institute:	The Maersk Mc-Kinney Moller Insitute
Education:	Civil Engineer in Software Engineering
Semester:	3
Module code:	T510013101
Project term:	2019-09-02 / 2019-12-16
Advisor:	Henrik Hoffmann
Group Nr.:	12

Group Members:

Frederik Alexander	Hounsvad	frhou18
Jakob	Gamborg Jørgensen	jajoe18
Thomas	Steenfeldt Laursen	tlaur18
Oliver Lind	Nordestgaard	olnor18
Nicolai Christian	Gram Rasmussen	nrasm11
Oliver Marco	van Komen	olvan18

DigID

1 Abstract

1.1 The Problem

The report problem proposition written within the constraints that the problem should result in a distributed system and contain aspects relevant from a cross cultural management perspective. With these constraint in mind, we worked towards finding a problem. As a result of our deliberation we ended up with the following problem statement:

How could one streamline identification, such that users do not disclose unnecessary and sensitive information and enable users to carry identification on their phone in a secure enough way for it to be recognized as legitimate.

This was the problem we found, which was incited by the annoyance of divulging unnecessary private information when showing one's id at times like buying smokes or alcohol, and by the poor and unsafe execution of existing solutions.

1.2 Approach

The report and software in the project were developed by way of scrum and unified process (UP). We chose to only do the first two phases of UP as the project only has academic purposes and as such only needs a functioning prototype and needs no deployment.

The first part of UP (inception phase) resulted in a set of requirements as well as an architectural design. The next phase (elaboration phase) started the development of prototypes, this included a backend server and mobile apps which resulted in a minimal viable product.

We chose to use the dynamic models as they had great value to our development process and chose not to create the static models as we saw no value to our development process.

We used scrum to structure the development of the project as well as the report. The structuring was done by taking the natural deadlines[1] and the deliverables and converting the tasks into sprints and thereby distributing the tasks on our scrum board hosted on Azure DevOps.

1.3 Results and conclusions

The result is a distributed software system with multiple applications running both on the client- and server-side. The prototype is fully functional but lacks a few non-crucial requirements due to low prioritization. The system proves that it is technically possible to make a secure identity verification system. In existing solutions, some security flaws were discovered. These flaws were fixed in the development of this system. Should the system roll out to international markets, some additional iterations of development are required.

[1] Natural deadlines being deadlines set by the university.

2 Foreword

The purpose of this report is to document the process and results of the development of a distributed system. More specifically the distributed system under development will be a high security id solution where identification is pulled from national servers to eliminate the need for trusting the client-side devices. The client-side devices will be phones or tablets for checking identification (Kiosk/Company users) and phones or tablets for allowing identification (Private users).

The project was started with a high ambition, and the scope of the project and the amount of work put into the project should reflect as such.

The report targets the project advisor, censor and fellow students. The technical level of the documents will therefore be at a level reflecting this.

We would like to thank some people who were instrumental to the project as the project of cause is rooted in an academic environment.

The group would like to thank:

- **Ronald Jabangwe** for coordinating the semester and giving us crucial knowledge on software architecture.
- **Mathias Neerup** and **Leon Bonde Larsen** for giving us essential knowledge on network infrastructure and containerisation which has been useful in the project.
- **Stine Berg** for the knowledge relevant for making development decisions in relation to deploying our solution in different cultures and countries.
- **Henrik Hoffman** for guiding us during our development process and inspiring us to make informed decisions as to what was important for us in our development process.

The group hereby recognise their own as well as the other members active participation in the project by their signature.

X [illegible]
Frederik Alexander Hounsvad

X [illegible]
Jakob Gamborg Jørgensen

X Thomas Steenfeldt
Thomas Steenfeldt Laursen

X [illegible]
Oliver Lind Nordestgaard

X Nicolai Gram
Nicolai Christian Gram Rasmussen

X [illegible]
Oliver Marco van Komen

3 Table of Contents

4 Editorial

Chapter	Responsible	Contribution by	Controlled by
Title page	**Frederik**		**All**
Abstract	**Frederik**	**Nicolai**	**All**
Foreword	**Frederik**		**All**
Introduction	**Jakob & Oliver N**	**Oliver Marco**	**All**
Problem analysis	**Jakob**		**All**
Methods	**Frederik**		**All**
Other solutions	**Oliver N**		**All**
Requirements	**Thomas**		**All**
Architecture	**Nicolai**		**All**
Design	**Oliver Marco**		**All**
Implementation	**Oliver Marco**		**All**
Security	**Oliver N**		**All**
Test	**Frederik**		**All**
Discussion	**Thomas**		**All**
Market analysis	**Jakob**	**Thomas**	**All**
Product Evaluation	**Oliver Marco**		**All**
Process Evaluation	**Thomas**		**All**
Conclusion	**Nicolai**		**All**

5 Figure overview

6 Introduction

As privacy becomes an increasingly important part of digitalisation of previously analogue systems, so should it for the non-digital systems already existing. One major place where privacy is currently not a focus is identification. When verifying age at a kiosk to purchase liquor or getting a package at the post office, one shows their ID-card, which usually has excess and often sensitive information that is unnecessary for the store clerk. In Denmark both the social security card, as well as the driver's license and passport show the social security number of the identification holder. The social security number is often used as the secret key for over the phone verification of identity in addition to easily accessible information such as address and name and could be used for sim swapping[2] or other kinds of identity theft. Therefore, our problem is:

How could one streamline identification, such that users do not disclose unnecessary and sensitive information and enable users to carry identification on their phone in a secure enough way for it to be recognized as legitimate?

Our solution is to create a system based on single-use tokens, where a client automatically generates a single-use token on a mobile app for the store clerk to scan with another app and request what information he needs to validate the client's identity. The client is then able to accept or deny the store clerks request for specifically what information they may need. This system additionally allows for a request that only checks whether a client is over a certain age, reducing information exposure and eliminating age calculating errors[3]. The system is based on the information from the government where available and photos are entered into the system at government service desks. This eliminates much of the fraud that can be found when using analogue identification, especially with age.

6.1 Background

Throughout this project we have been using unified process and Scrum which is a use case driven iterative process. A use case is a diagram showing a user's interaction with the system. Scrum is beneficial for the project because its focus on the agile process combined with strong project management. Unified process also known has UP is an object orientated software development process and combined with Scrum it ensures we work on the project incremental.

We have been using technologies such as ASP.NET Core 3.0, which is a webserver framework written in C#. We have been using flask for a couple of small services. Flask is a python framework, for making simple API's. The previous two mentioned frameworks have been used for our backend.

For our frontend we have been using VueJS, a Javascript framework, and Android studio, for the making of two android applications. To increase the security for the system, we have containerized our different project using Docker, and secured the backend by way of a reverse proxy.

[2] The practice of taking control over a targets phone number usually with the purpose of exploiting password resetting as a text message.
[3] It is not uncommon for supermarket kiosk to have pieces of paper with birth year to common ages marked down, seeing as many cannot reliably calculate an age based on the birth date.

7 Problem analysis

Showing your identification card, not only shows your name and your birthday, which is the information the store clerk needs, it also shows your social security number. This is a problem as your social security number is sensitive information, and in the hands of the wrong people, this can be used for fraud.

When you are buying alcohol or cigarettes in a store, you are not forced to show your CPR-number (Bergeon, 2018). But the store clerk, will ask for your identification, as they are required to verify your age. Is this the store clerk breaking the law, or is this a mistake in society?

When people are to identify themselves, they will in the most cases automatically take up their driver's license and show it to the store clerk. Even though the problem in given out your CPR-number is a well-known issue. But what are they to do? They need to identify themselves, and the most common way to do this, is by showing their driver's license. This is the way the government have offered us to do it and taking these things into consideration you could ask yourself: "Should the government find a better solution for this issue?"

8 Methods

The initial processes conducted during the production of our solution were the initial idea generation, where ideas were simply brainstormed and then singled down by process of elimination.
The analysis of the problem space came next where the group investigated existing solutions, both to gather useful ideas as well as to see what may be wrong with existing solutions, and where they might simply be lacking, but also to see what may have been done right in those implementations.
Then the analysis of the requirements came. By looking at what we had found in the analysis of problem space we could draw out the different use cases for our program, thereby locking down the functionality of our program.
From the use cases we could have made an analysis model, but after several discussions we came to the conclusion that an analysis class diagram would be unnecessary, and unhelpful for us in the process of designing our system, and as such the static analysis was eliminated from the model. The dynamic part of the model on the other hand proved to be extremely useful, both to lock down the functionality, but also to understand each other's different takes on the flow of the different use cases.

Knowledge gathering has been an important point throughout the inception and development process. We have gathered knowledge from various sources depending on the subject. Most of our knowledge gathering has been done thru the various vendor's own documentation and thru articles to help us analyse our problem, but we have also acquired a plethora of knowledge from the different courses, such as the use of a reverse proxy for security.

Unified Process has been utilised to some degree in the project. We have only utilised the first two parts of UP as the project is an academic endeavour and therefore has no rollout and no maintenance. The use of UP was also encouraged by the scheduling of sub deliveries from the subject coordinators, as this clearly segmented the process by having requirements and other artefacts fitting into the inception phase turned into a deliverable with a clear deadline.

Coordination of work can be a tricky subject as it really depends on the work being done. We decided to utilize scrum as we are working in an agile fashion according to UP.

We ordered our work schedule and work items in sprints and backlogs as were to be working under the structure that is scrum. To coordinate such a process as scrum can be done in many ways, but we decided to work with Azure DevOps[4] as it would provide hosting of code, documentation and also coordination of sprints and backlogs of work items in an ordered and neat fashion. The lengths of our sprints were based on whatever the specific task were and what the deadlines for specific functionalities and artefacts were, but the sprints ranged from a single week to three.

Our scrum was structured with a few scrum buts:

- Scrum but weekly scrum instead of daily scrum
- Scrum but no product owner

The process of writing the code was done in various ways. A large amount of the code was written by way of pair programming, as it increases both the quality and the speed at which the software is produced. This pair programming was also interspersed with spouts of individual programming to solve smaller problems or when the time did not allow for waiting for schedules to align.

[4] A product available by way of Microsoft.

9 Other solutions

Having looked for other solutions, we found one created by, amongst others, Danske Spil, called Smart ID, which has most of the lower level features of our system, meaning verifying age and identity by way on an image. They did, however, make various design choices that have given their system a lot of issues, and we therefore decided to learn from their mistakes and successes in order to better solve the problem of secure and trustworthy digital identification. We have found no other solution in our target market which solves or aims to solve the same problem of identification.

The first and most glaring mistake of their system, a newspaper, as well as their Play Store comment, have called them out on, is the user selected images (Visbøll, 2017). As there is no verification of the selected image, anyone over 18 could login to with their NemID on a minor's app, and then take a picture of said minor, making them over 18 for all the system and the kiosk employees knows. SmartID could have solved this issue by having a verification requirement using the Danske Spil kiosks that are already very common, to verify the images uploaded is of the actual person.

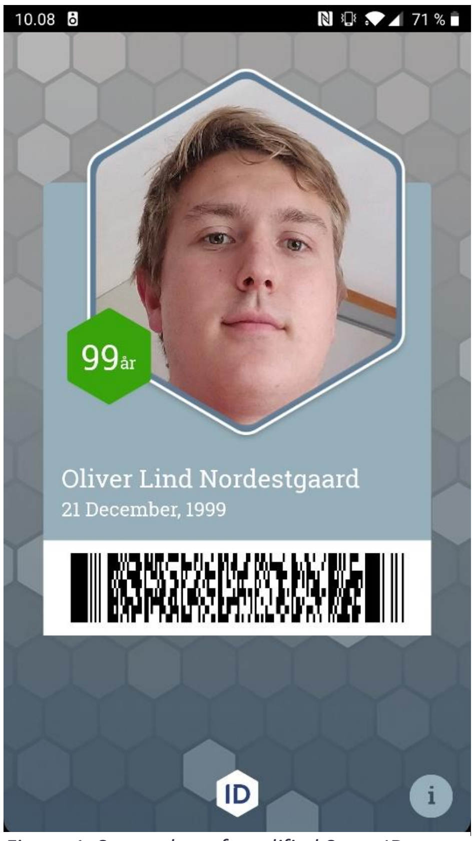


Figure 1: Screenshot of modified SmartID app

Another mistake of the SmartID system, is the fact that whilst it uses server-authoritative architecture it still has a client-side version and view of the information. This is bad as it enables users to modify their application in such a way that the information shown on their app is false and only proven false by the scanning at a Danske Spil terminal. That would mean during rush-hours when the queue terminal is long, many employees would skip the verification scan, and assume that the information presented by the application is valid. Furthermore, in a small questionnaire[5], we found that none of the kiosk employees, including the boss at a specific location even knew that the applications information could be verified at all. The android app proved to be quite simple to tamper with as all the obfuscated variables had json tags in clear text, meaning that the variable "b48d" suddenly made a lot more sense as "age". We proved this by changing the app to only display the users age as 99 years old. From this we learned, that if we chose not to display any user information on the app, the possibility of just trusting the client application and not verifying it, is non-existent. It does mean that out system cannot be used as identification between two private users, but that was an acceptable drawback if it means that the system forces the employees in charge of age verification to do the work properly and not cutting corners.

[5] The questionnaire was done verbally at one of the group members' workplace, which was a kiosk selling Danske Spil games.

10 Requirements

When developing a system, it is important to understand exactly what needs to be done. This knowledge is achieved through defining requirements. In this section, we will go through the scope of the system and describe what it should be able to do through use cases. The most important requirements will be described in detail.

10.1 System scope

To solve the main problem, the product of this project needs to consist of several different pieces of software. A mobile application to be used by the private users is needed. This app should allow the users to sign in preferably with NemID. Afterwards, the app shall show a personalized QR code for the clerks/ID controllers to scan. These clerks/ID controllers will use a different mobile application to scan the QR codes and retrieve information about the private users. You could combine these functionalities in one single app; however, this would make the solution more complex than necessary, as private users would never use the ID controller-functionality, and ID controllers would never use the private user functionality.

The app used by store clerks needs to be configured somehow to fit the company it operates in. Some places need to know people's ages and others need to know their addresses etc. This will be configured by company admins through a website where the admin can choose what information the clerks are able to scan off the customer's apps. Furthermore, DigID will also need an Internal Client for the internal stakeholders to use. Here, new companies can be added to the system.

These applications need to communicate backend server. This server should handle almost everything by retrieving requests from the different clients. It should then answer back to them with appropriate responses. The server will also need a connection to a database where data is stored to and retrieved from.

If this was a real-world project, the system would need to communicate with NemID to retrieve the social security numbers of its users. As this would be impractical in this project, we need to construct a replica of the NemID system to imitate how DigID would look like in the real world. The same applies for the "CPR register"-part of the system. In the real world, DigID would use the CPR register to look up social security numbers to retrieve information about the people in question.

Because this system will handle the personal data of people, the private users will need to be registered in a way that is recognized by the government. The idea is that citizens can request to be registered at Borgerservice. The Borgerservice employees will then handle the registration through a website. Therefore, we will also need to develop a Borgerservice client to allow for employees to create, delete and alteration of private users in the DigID system.

Ideally, we want to extend our solution to enable use by third party developers. This would be useful when someone decides they want to use our identification system in their own way. For example, you could use this identification system to unlock doors at a facility instead of making employees carry identification cards.

Figure 2 shows the DigID deployment diagram. This diagram gives an overview of how the different components of DigID will communicate with each other, and how DigID should be able to communicate with outside entities like NemID and the CPR register. Here, it is easily seen how the backend server acts like a gathering point for all communication in DigID.

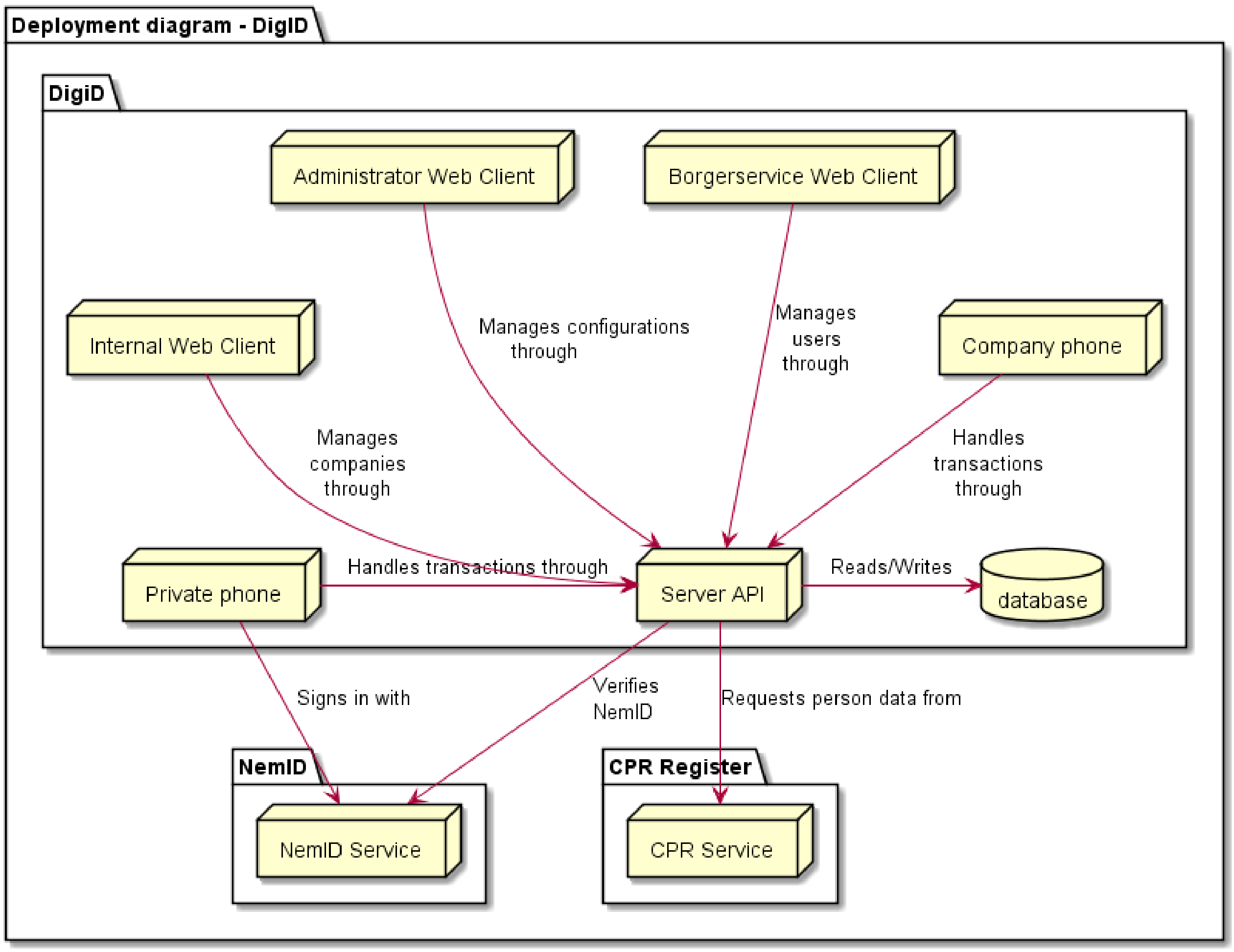


Figure 2: Deployment Diagram

10.2 General requirements specification

10.2.1 Use case diagrams

Several of DigID's components are used by primary actors. The interactable components of DigID are the following:

- Private Client
- Company Client
- Administrator Client
- Borgerservice Client
- Internal Client

The Private Client represents the mobile application used by the private users. The Company Client is the mobile application used by the ID controllers/clerks. Company administrators use the Administrator Client website to manage the ID controllers. Borgerservice Client is the website where Borgerservice employees manage the private users. The Internal Client represents the website where the internal stakeholders manage the companies that use DigID.

Each separate interactable component has a use case diagram. These can be found in the Appendix. A component like the backend server is not to be used directly by any actors. Therefore, it does not have any use cases. However, it is still required to fulfil the needs of the components that communicates with it. Several different actors use DigID. The table below contains descriptions of them all.

Actor list	
Internal administrator	An internal administrator is a person who manages all companies (customers) of the system. In practice this would be the people working on the project, and likely the Danish government.
ID controller	An employee of a business that uses DigID. This could be the clerk at a kiosk or a bouncer at a bar. They are able to use DigID to validate customers' age and identification.
Private user	A customer of a business that uses DigID. This customer can get their ID scanned off their DigID app to prove their age or identification.
Company Account Administrator	The administrator is accountable for configuring the id checkers accounts and app layout
Borgerservice	Borgerservice is responsible for managing private users of the system. They should be able to register photographs for private users, alter private user photos & remove private users.
Private user	A customer of a business that uses DigID. This customer can get their ID scanned off their DigID app to prove their age or identification.
Server	The DigID backend API. This is the backend that handles the business logic of DigID. All the clients make requests to this.
NemID	The NemID service is an aspect of the real world that will be replicated in this project. This system allows for private users to sign into the private user app when registered at the Borgerservice.
CPR-register	The CPR-register in an aspect of the real world that will be replicated in this project. This system contains personal information about every citizen. DigID uses the CPR-register to get information such as age, date of birth, address and pictures from a CPR number.
API Integrator	A person/company can integrate DigID in their own system where they will be able to request users ID and get users ID without the use of our company APP. The integrator will get a private API key which they will be able to use to make requests to our server.

Not all components were prioritised equally in this project. The table below illustrates the priority of each component and the priorities of each of the components' requirements. The priorities are described using the MoSCoW method.

DigID component	Compo-nent priority	Require-ment ID	Requirement	Require-ment prior-ity
Private Client	M	PC-1	Sign in	M
		PC-2	Unlock	S
		PC-3	Start transaction	M
Company Client	M	CC-1	Sign in	M
		CC-2	Unlock	S
		CC-3	Request ID	M
Administrator Client	M	AC-1	Sign in	M
		AC-2	Create ID controller account	S
		AC-3	Alter ID controller account	S
		AC-4	Remove ID controller account	S
		AC-5	Adjust App configuration	M
		AC-6	Get statistics	C
Borgerservice Client	S	BC-1	Sign in	M
		BC-2	Register photograph for private user	M
		BC-3	Delete private user	M
Internal Client	S	IC-1	Sign In	M
		IC-2	Setup new company	M
		IC-3	Remove company	M
API	C	AI-1	Make API call	C

10.2.2 Non-functional requirements

The table below describes the non-functional requirements related to DigID.

ID	Description	MoS-CoW
F00	The system must not take long to open and be ready for a transaction to happen	S
F01	The system must not take long to handle a transaction between a company and a user	S
F02	The system must ensure stored data is unreadable by unauthorized people, for example by encrypting	M
F03	The system must make it difficult for malicious users to hack others, for example by limiting the amount of login tries	S
F04	Users need to have their picture taken at an authorized place	M
F05	The servers must be able to handle at least 5% of all registered users at the same time	S
F06	Citizens should not be allowed to change any of their information with the application	M

F07	The mobile applications must work on Android	M
F08	Users must accept terms of service before using the application	M
F09	The system must be distributed	M
F10	The Mobile Application must work with slow internet connection	S
F11	The Mobile Application must be energy efficient	C
F12	The server must be developed with a modular design for 3rd party integration and expansion	C

10.2.3 Domain model

Figure 3 below illustrates the domain model for DigID. When an interaction between a store clerk (Company Client) and a customer (Private Client) happens, the store clerk will scan a QR Code on the customer's app, which will start at transaction which communicates with CPRregister and sends data back to the store clerk which will inform the store clerk if the citizen is old enough to buy a pack of cigarettes for example. DigID makes requests to the CPRregister with every scan for two reasons. First; it will ensure the retrieval of up to date data. Second; otherwise we would have to store people's personal data in our database making it less safe than if it was stored only one place. In this project, we are not using the real CPRregister but a self-made replica. However, if this was a real-world project, it would be possible to make requests to the real CPR-register (Det Centrale Personregister, 2019).

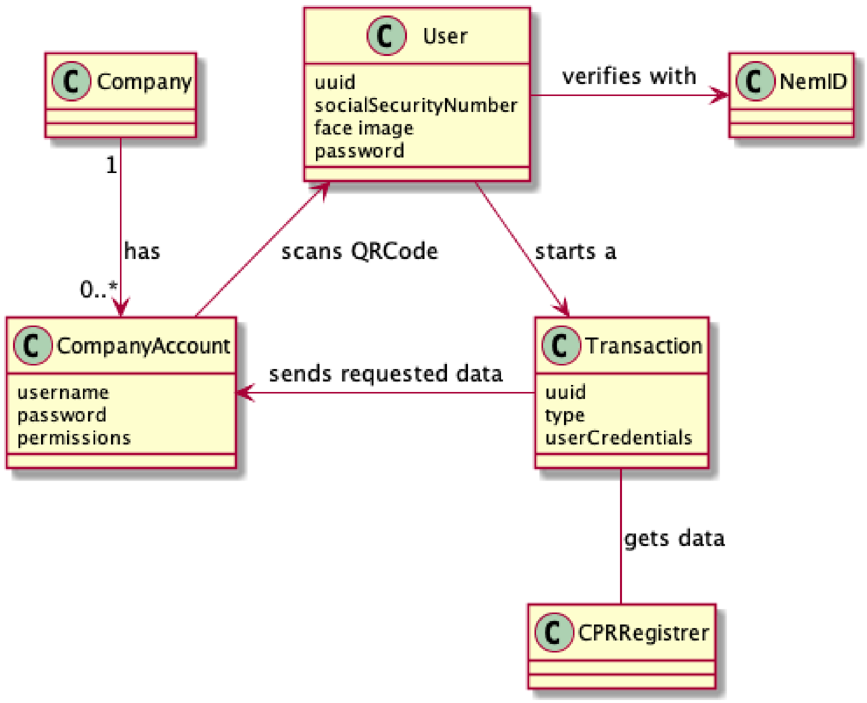


Figure 3: Domain class diagram

10.3 Detailed requirements specification

In this section, the most important requirements are described in detail to help understand precisely how the different aspects of the system are to work. The most critical requirement in this product, is the requirement to make and complete transactions between a private user and an ID controller. Therefore, requirement PC-3 and CC-3 is described in detail.

Start Transaction

ID: PC-3

Primary actors:
Private user

Secondary actors:
Server, ID Controller

Short description:
The user wants to verify their age or identity

Preconditions:
1. The use case "ID: PC-2"

Main flow:
1. The client queries the server for a transaction ID.
2. This ID is presented as a QR code on the apps home screen.
3. The user then presents this code to the ID Controller.
4. When the ID Controller has scanned the QR code and selected what he wants to verify, the user is promoted with what the ID Controller requested and asked to confirm this request
5. **If** the user agrees
   a. The transaction is completed
6. **Else** the user does not agree
   a. The request is denied
   b. The transaction is terminated
   c. The user is presented with an option for a new transaction id

Postconditions:
1. The verification is completed.

Alternative flows:
2. Connection error
   a. This flow starts when the user client tries to start a transaction with no connection to the relevant servers.
   b. The user Client displays an error.
   c. The user is free to try again when they have a connection.
3. No one scans the code.
   a. The transaction is started but no-one scans it.
   b. The transaction ends after 15 minutes of inactivity
   c. The user is presented with an option for a new transaction id

Request ID
ID: CC-3
Primary actors: ID controller
Secondary actors: Private user, CPR register
Short description: The ID controller can scan private users’ ID to verify their age, identity etc.
Preconditions: 1. The ID controller is at the home page of the Company Client.
Main flow: 1. The ID controller chooses from a list of verification types (age, identity etc.) 2. The ID controller uses the camera to scan the private user´s QR code 3. **If** the private user denies the request a. The company client is presented with a message saying the user denied the request 4. **Else** the private user confirms the request a. The server requests the relevant information from the CPR Register b. The company client displays requested information
Postconditions: 1. The home screen is displayed in a ready state
Alternative flows: 1. Connection error a. This flow starts if the Company Client loses connection to the server anytime during the main flow. b. The Company Client displays an error message and returns to the home page.

11 Architecture

This section presents high-level architectural decisions made while developing the system. The architecture of the system was decided in the early stages of the project and has proved to last. After gathering the requirements, the group has been clear on what architecture was required to build the system.

11.1 Client-Server

When a private person uses the application to claim that he is over 18 years old, we cannot trust the client application to give us the correct information, as the application could potentially be modified by a technically advanced user.

This is the reason why the system relies on a client-server architecture, where the server acts as the only source of truth and needs to validate actions done in the system.

Figure 4 below is a representation of the client and the server which shows examples of each of their responsibilities.

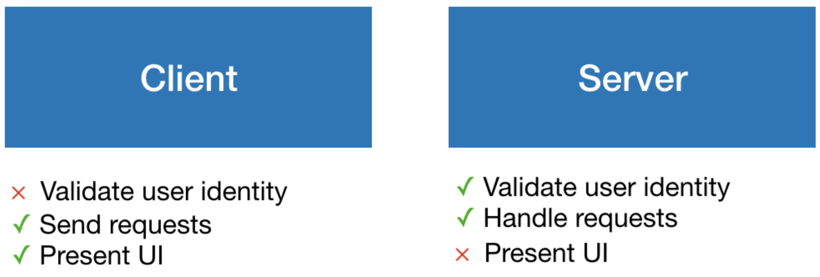


Figure 4: Client Server Responsibilities

11.2 Distributed Systems using Web API

The system needs to support multiple types of client applications (Android App, Web App and potentially more). These applications are distributed on to the client's computer/phone rather than being served by a server. In contrast, traditional web-apps using the MVC pattern a backend application would be responsible for serving the UI (html files) that could be used to call methods on the server. However, such a setup would not fit this project due to having multiple clients. Instead the backend application will function as a Web API that is responsible for reading and writing from/to the database.

As seen in Figure 5 below the applications will use JSON as a common format to communicate between each other, thus making the underlying programming language or framework insignificant. Every language used should be able to parse the data it is manipulating into JSON format and back. This will be done with either framework-native functions, or with third party libraries, depending on the individual implementation.

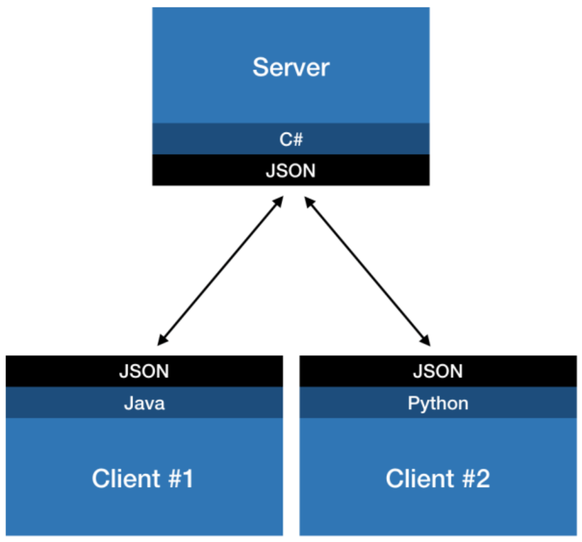


Figure 5: Json Communication Illustration

However, as JSON is only a data format, it is not enough to form a contract between applications. There also needs to be a way to transmit the data. This is done using HTTP requests following the REST convention (see 10.4)

11.3 Microservices

For the whole system to work, multiple applications need to be running. This is because they are separated into microservices.

There are two stateless services running besides the primary backend. One microservice is the CPR register, and another is the NemId system. These can run independent of the other services that use them.

Microservices has some pros and cons, as highlighted in the article (Monolithic vs. Microservices Architecture). Microservices initially require a bit more setup and configuration, as well as more maintenance as compared to a monolithic application. On the bright side it provides a service entirely decoupled from other code in the system. Therefore, a microservice isn't coupled to any constraints of the main application such as programming language, framework, or even hardware constraints such as operating system, CPU and so on. Likewise, the service can be deployed independently which makes it easier for a team to work on and update a specific service.

11.4 Representational State Transfer (REST) and RESTful services

When a system is distributed across multiple application using different programming languages and frameworks, a common way to transmit data is required. In this project, it is achieved by sending JSON via HTTP requests using the REST convention between the applications.

REST provides interoperability between the system interfaces meaning that any application will be able to communicate with any other regardless of technological constraints such as language or framework. This is a very important point considering that the system must run on multiple platforms.

For a service to be RESTful, the requests are required to be stateless. This makes it much easier to debug and test the application, because every time you make a request, you can expect the same type of response. The data might be different, but the request will be handled the same exact way every time.

11.5 Containerization

As applications split up into multiple smaller projects, deployment becomes a heavier process. Containerization is an architecture that attempts to solve this problem by encapsulating applications inside smaller containers, which allows easy deployment and scaling by letting a developer run an application inside of a production environment on their local machine.

Every service of this project is bundled into a container. To orchestrate these containers, docker-compose is used to run them all at once, so you can ensure that all required services are running and are able to communicate with each other.

11.6 Physical Architecture (3-tier and microservices)

Here's an overview of the applications which are all deployed separately.

The main flow of the system is built around a 3-tier architecture, where the backend is responsible handling calls from the client applications and updating the database. All the applications have access to the separate services CPR Service and NemID service.

The simple deployment diagram below (Figure 6) explains all the physical services running when the system is running. It also gives an overview of how the applications communicate with each other. As it shows, most actions are done on the backend server, and the microservices are available to all the applications.

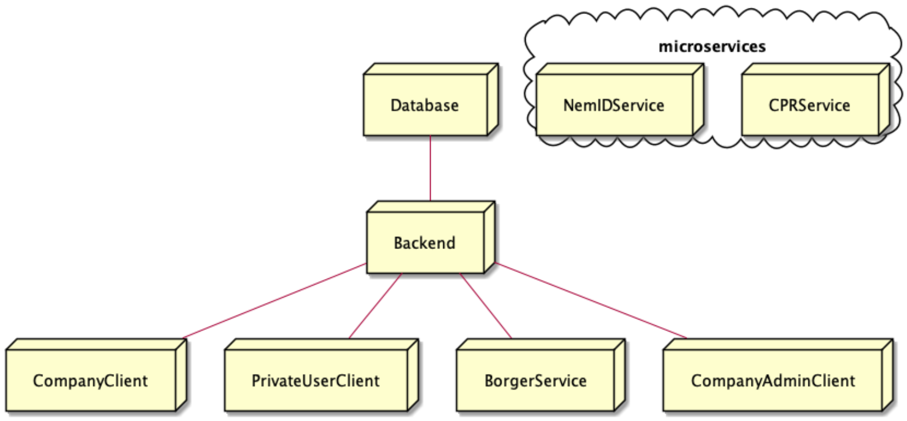


Figure 6: Simple Deployment diagram

Further details of software running on each service:

- Database
    - PostgreSQL
- Backend
    - .NET Core 3 Web API
- CPRService
    - Flask API
- NemIDService
    - Flask API
- CompanyClient
    - Android App
- PrivateUserClient
    - Android App
- CompanyClient
    - VueJS Web App
- Borgerservice
    - VueJS Web App

12 Design

Note: All diagrams in this chapter has been made as a way to document the system after development was done. These diagrams have not been made or used during development as the group decided against this approach. The system has been developed based on analysis sequence diagrams and group/pair discussions.

This section focuses primarily on the server, as this is where all business logic resides. Types will only be shown once in the following diagrams. This means when a variable 'password' is shown for the first time in a diagram, it will have a type of string. If the variable is used again in another method call, the type will be left out for condensation.

All the diagrams have left out communication between the server and database. There is a section in the chapter explaining in detail how the server communicates with the database.

12.1 Broad system overview

Figure 7 below shows the most important aspect of DigID. It shows how a private user and a controller communicates with the server through transaction controller and how use case Start Transaction ID: PC-3 and use case Request ID ID: CC-3 is realised.

A user opens their personal app, the app sends their NemID token and hardware ID to the server in order to request and receive a transaction ID. The transaction ID is used to generate a QR code which an ID Controller scans with their app. The user then gets prompted whether or not they want to share the information which the ID Controller has requested. They will either accept or deny this request. If the user accepts, the ID Controller will receive the information they requested, if they deny the ID Controller will be able to scan again.

Figure 7 assumes the user is already logged in. Later in this chapter it will be explained how the user logs in.

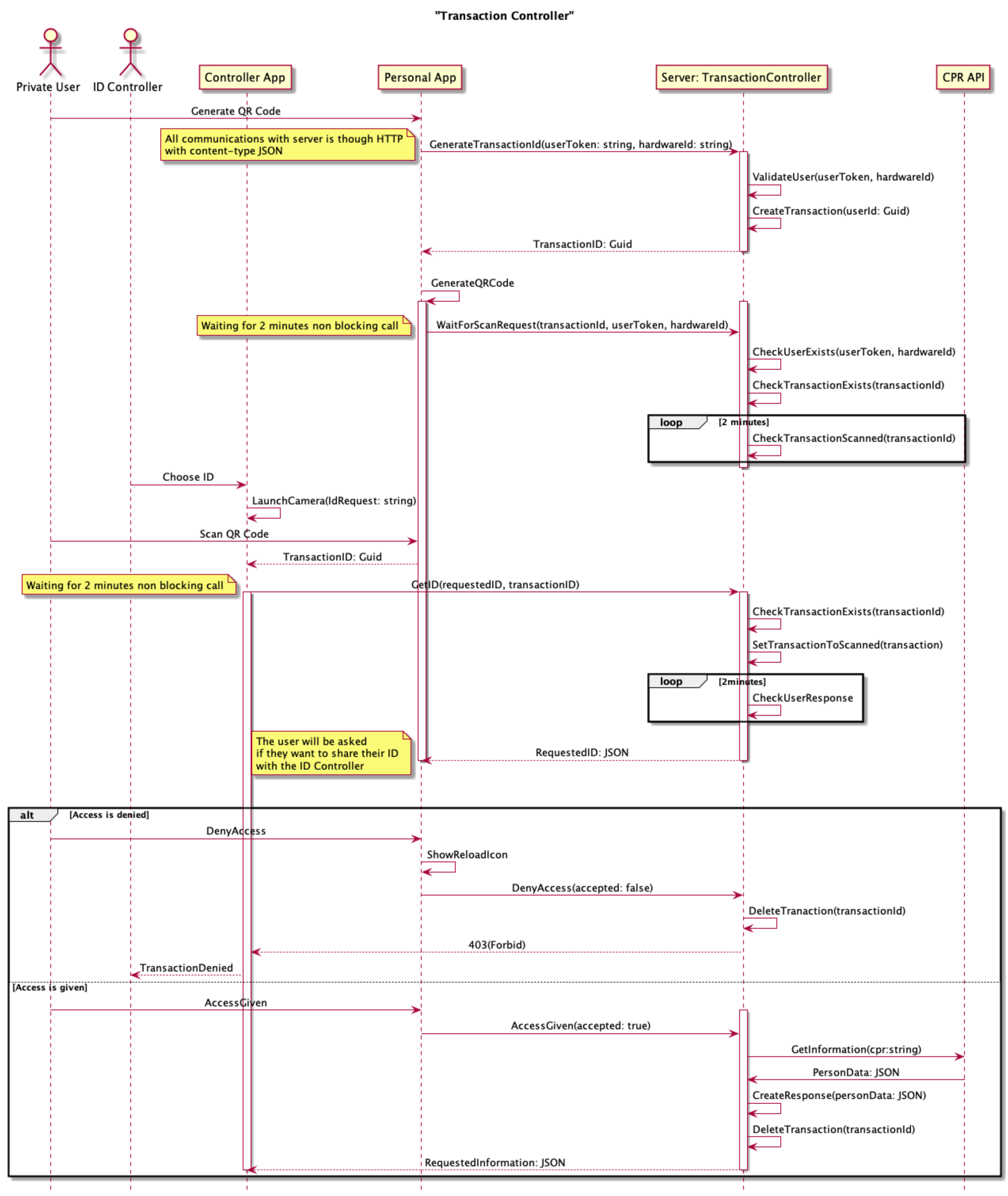

"Transaction Controller"
Private User
ID Controller
Controller App
Personal App
Server: TransactionController
CPR API
Generate QR Code
All communications with server is though HTTP with content-type JSON
GenerateTransactionId(userToken: string, hardwareId: string)
ValidateUser(userToken, hardwareId)
CreateTransaction(userId: Guid)
TransactionID: Guid
GenerateQRCode
Waiting for 2 minutes non blocking call
WaitForScanRequest(transactionId, userToken, hardwareId)
CheckUserExists(userToken, hardwareId)
CheckTransactionExists(transactionId)
loop
[2 minutes]
CheckTransactionScanned(transactionId)
Choose ID
LaunchCamera(IdRequest: string)
Scan QR Code
TransactionID: Guid
Waiting for 2 minutes non blocking call
GetID(requestedID, transactionID)
CheckTransactionExists(transactionId)
SetTransactionToScanned(transaction)
loop
[2minutes]
CheckUserResponse
The user will be asked if they want to share their ID with the ID Controller
RequestedID: JSON
alt
[Access is denied]
DenyAccess
ShowReloadIcon
DenyAccess(accepted: false)
DeleteTranaction(transactionId)
403(Forbid)
TransactionDenied
[Access is given]
AccessGiven
AccessGiven(accepted: true)
GetInformation(cpr:string)
PersonData: JSON
CreateResponse(personData: JSON)
DeleteTransaction(transactionId)
RequestedInformation: JSON

Figure 7: Transaction Control Flow

12.2 Authentication

There is two different ways to authenticate and validate users. For administrators, IdControllers and borgerservice accounts JSON web token (JWT) is used. JWT is a standard made by the Internet Engineering Task Force (IETF). It is a compact and secure way to transfer claims between two systems. A claim is a piece of information, with a key and a value with a JSON format. The information in these claims can be trusted, because JWT is signed. A JWT consists of three parts separated by a dot, aaaaa.bbbbb.ccccc. The first segment is a header which describes the algorithm being used, and token type. Example: {"alg":"HS256", "typ": "JWT"}. The second segment is the payload which consists of claims. Example: {"userId": "1", "name": "Peter"}. The third segment is the signature which takes the header and the payload and signs them with a secret using an HMAC algorithm. (JWT.io, u.d.)

JWT in our system uses SHA256. The payload consists of username, userId, accountType, nbf (not before), exp (expiration time), iat (issued at). Where nbf, exp and iat is timestamps. Nbf is useful to avoid brute force attacks, exp is to avoid users being able to log in forever with the same token, and iat is creation time.

JWT is not used for private users. Private users' identity is validated using a NemIdToken and hardware ID. Whenever a private user makes a request, they send a NemIdToken which they receive the first time they logged in with NemID, and hardware ID unique to their phone. These two strings are then validated and if the system recognises them, the user is able to make the request.

12.2.1 Private user authentication

Figure 8 below shows the authentication flow for a normal user. In order to use the app, they must be NemID verified. The first time they log into the app a NemID form is loaded, which will create a NemID token after a successful login.

The token will be used alongside a hardware ID, unique to their phone, to verify their identity on the server at PrivateUserController. When they have verified with NemID, they are no longer required to authenticate with NemID, they will instead be able to choose a PIN.

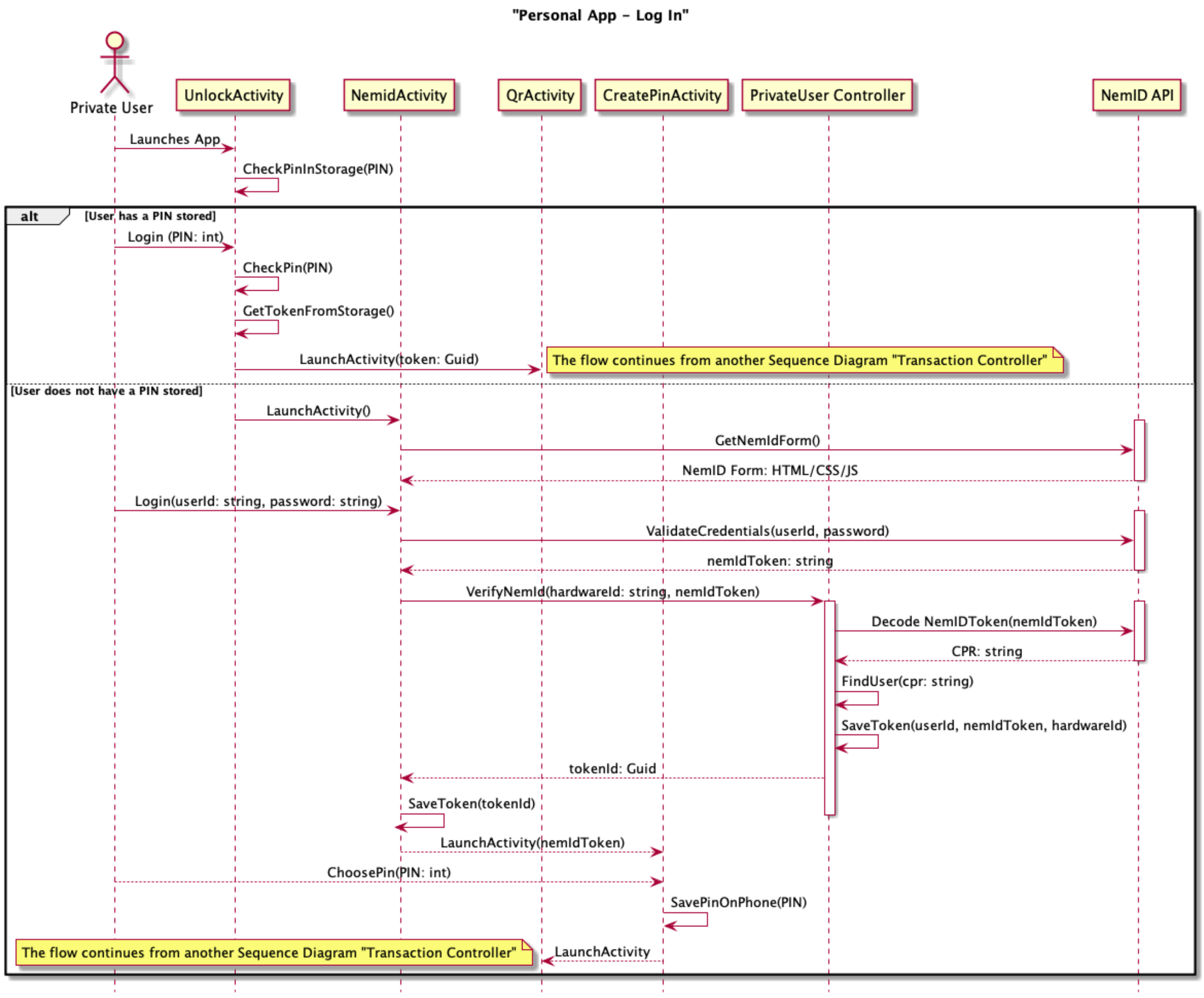

"Personal App – Log In"
Private User
UnlockActivity
NemidActivity
QrActivity
CreatePinActivity
PrivateUser Controller
NemID API
Launches App
CheckPinInStorage(PIN)
alt
[User has a PIN stored]
Login (PIN: int)
CheckPin(PIN)
GetTokenFromStorage()
LaunchActivity(token: Guid)
The flow continues from another Sequence Diagram "Transaction Controller"
[User does not have a PIN stored]
LaunchActivity()
GetNemIdForm()
NemID Form: HTML/CSS/JS
Login(userId: string, password: string)
ValidateCredentials(userId, password)
nemIdToken: string
VerifyNemId(hardwareId: string, nemIdToken)
Decode NemIDToken(nemIdToken)
CPR: string
FindUser(cpr: string)
SaveToken(userId, nemIdToken, hardwareId)
tokenId: Guid
SaveToken(tokenId)
LaunchActivity(nemIdToken)
ChoosePin(PIN: int)
SavePinOnPhone(PIN)
The flow continues from another Sequence Diagram "Transaction Controller"
LaunchActivity

Figure 8: Log In control Flow

Figure 8 above shows 3 activities, UnlockActivity, NemidActivity and QRActivity, which areq android classes in charge of the view. These classes coordinate business logic, such as communicating with the server and NemID API.

12.2.2 ID Controller authentication

For an ID Controller to be able to log in, they have to be registered. This is done by their Administrator. When a store uses DigID, they will get an administrator account which will be able to create new ID Controllers accounts. When they create an ID Controller account, they enter credentials in the form of username and a password which the ID Controller will use to log into their app. The password is then hashed and salted with their username before saved to the database.

Figure 9 below illustrates how an ID Controller logs into the company app. They enter their credentials, which is hashed and then posted to AuthController, which will check their username, hash the password and check the hashed password up against the database. If the credentials match, the server generates a json web token (JWT) and sends it back. This JWT holds their username, server ID and account type.

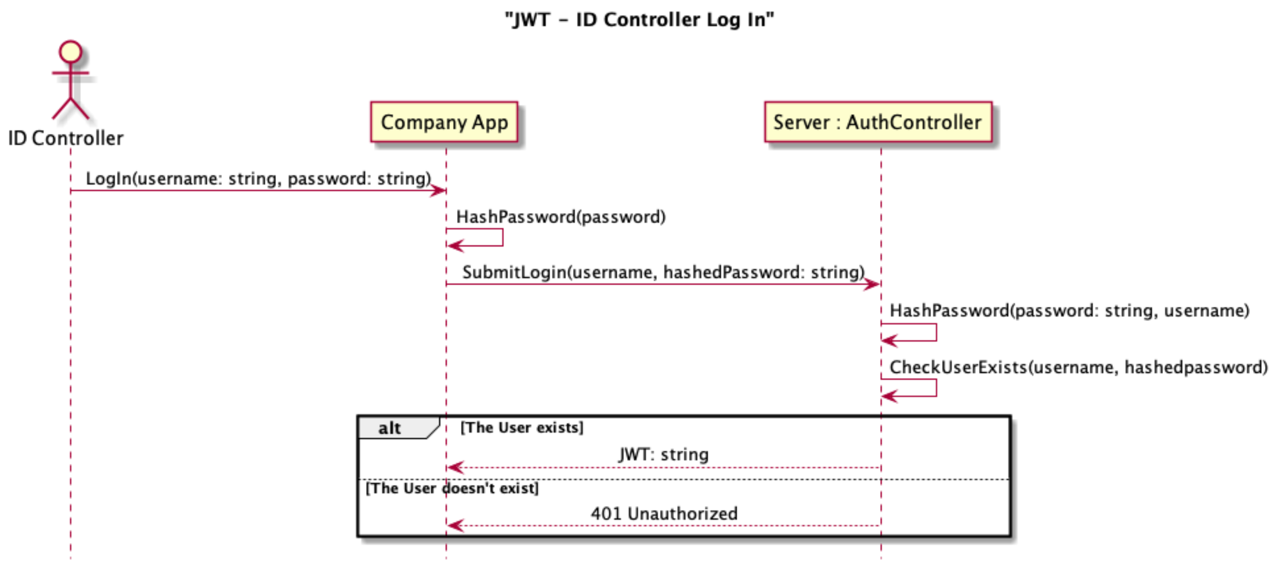

"JWT – ID Controller Log In"
ID Controller
Company App
Server : AuthController
LogIn(username: string, password: string)
HashPassword(password)
SubmitLogin(username, hashedPassword: string)
HashPassword(password: string, username)
CheckUserExists(username, hashedpassword)
alt
[The User exists]
JWT: string
[The User doesn't exist]
401 Unauthorized

Figure 9: Log into company app

The JWT is valid for 2 hours. Each time the company app communicates with the server, the JWT is validated and used to validate the user id (the id controller who is making a request) and the account type. The controllers will therefore be able to forbid or accept request based on the JWT and the Id Controller won't need to have to login every time the app has been closed.

12.3 Database access

This section explains which tables and entities our database holds and how the database is managed.

12.3.1 Entities overview

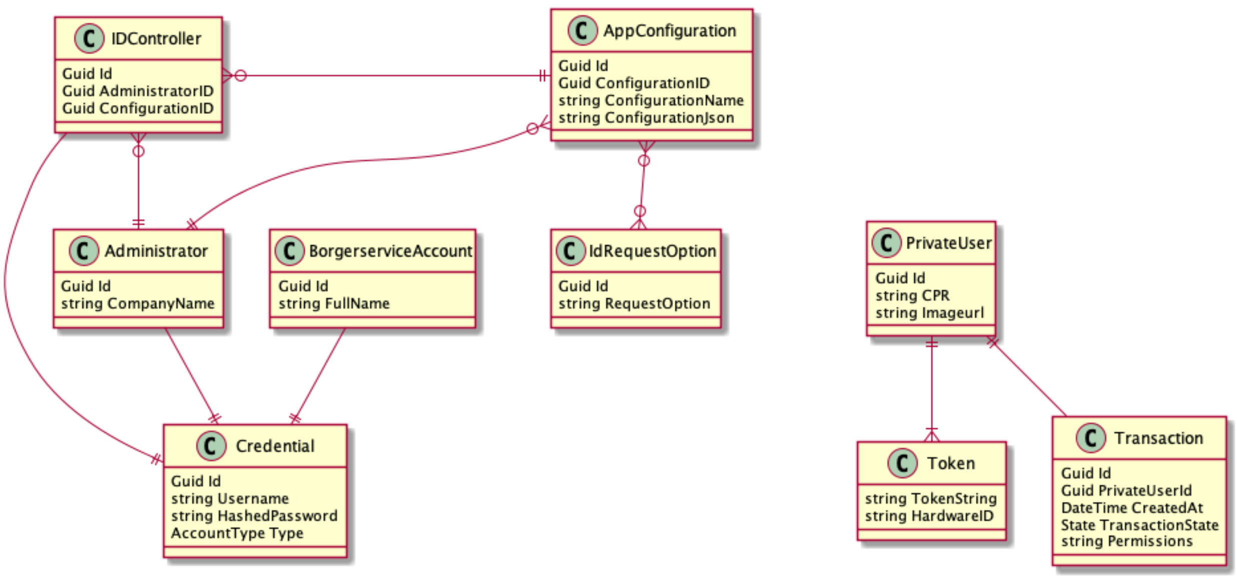


Figure 10: Entity overview

In Figure 10 database entities are shown.

- **IDController** needs to have exactly one AppConfiguration, one Administrator and one credential.
- **Administrator** can have zero to many AppConfigurations, exactly one Credential and zero to many IdControllers.
- **BorgerserviceAccount** needs to have exactly one Credential.
- **AppConfiguration** needs to have exactly one Administrator and has zero to many IdRequestOption. An IdRequestOption could, among many others, be “Birthday”, “Over 16 years old” etc.
- **IdRequestOption** has zero to many AppConfigurations.
- **PrivateUser** can have one to many token and a token can only have one user.
- **Transaction** can have exactly one privateUser.

12.3.2 Entity Framework Core

Entity Framework Core (EF Core) is an ORM (Object-relational mapper) which makes it easier to work with and access database objects. It enables the developer to work directly with C# entity classes, without having to write much SQL and using the power of LINQ. It saves the developer a lot of manual work and time. In this project a Code-First approach has been used, which enables developers to define C# Classes with properties and EF Core specific annotations and create a database based on these classes with migrations.

Migrations is a way to change database tables and structure over time, so if an entity class needs to have a new property, it can be added to the class, a migration can be generated, and the database can then be updated based on the latest migration.

EF Core uses a lot of conventions, so if an entity class has a property called Id, EF Core knows that this should be an auto generated primary key field in the table. There are also specific ways to configure entity classes to properly configure one to many, and many to many relationships.

EF Core can be used with many different SQL languages, in this project EF Core has been used with PostgreSQL.

12.4 Controller structure

All of the controllers on the server share common characteristics.
They all use automapper, DTO’s (Data Transfer Objects), repository pattern and most of them use JWT Authorization with custom policies.

12.4.1 DTO’s

Routes will specify the values it requires. Modelstate will then verify that these fields exist, and validate that they are of the correct types, which means it will automatically convert a string in json format to a real type which the C# compiler understands.

As an example: The POST route AuthController (SubmitLogin) requires a username and a password. Which the route will receive through a DTO called LoginUserDTO. If the user doesn’t send a username or password, they will automatically receive a 400 bad request.

12.4.2 Automapper

Automapper is a NuGet package which saves a lot of manual work when a new object has to be created based on another object’s values, it transfers values from one object to another.
As an example: When an ID Controller is created at POST route ClerkController, the administrator creating the ID Controller has to specify some of the values that goes into creating a new Id Controller account but, some of the fields such as Account Type is set by the server itself. Therefore, when a new Id Controller object needs to be made some of the fields should be transferred from the received values from the user input. Instead of creating a new ID Controller object and manually setting the fields.

12.4.3 Repository pattern

Figure 11 below shows how controllers access the database. All controllers will get an appropriate interface injected with dependency injection, so that they aren't coupled to a concrete implementation. This interface is a repository which has relevant methods for communicating with the database. These repositories use an ORM (Object relational mapping) framework, Entity Framework.

The benefits of this pattern are many. The most important for us is following SOLID principles. If we decide to scrap Entity Framework and use Raw SQL communication instead or another ORM, the controllers will not be affected by this change. The controllers and repositories focus on a single responsibility and they rely on an interface which gets injected. Other benefits include easier unit testing of the controllers. When the controllers are unit tested the repositories can be mocked and injected without the use of a database connection unit tests can continue to be fast.

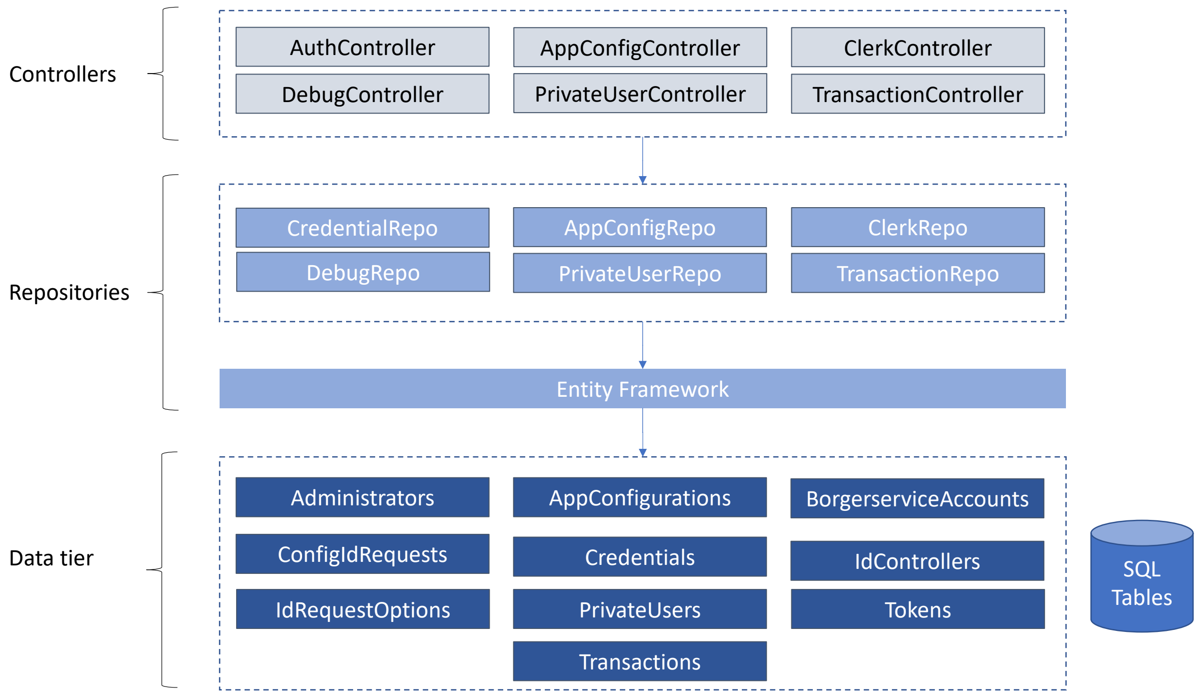


Figure 11: Backend architectural overview

12.5 JWT Authorization & policies

Before a request hits an endpoint in a controller which requires authorization, the JWT will have been validated successfully. Furthermore, a controller can describe policies, which could require specific attributes and values the token should have.

As an example: Only an administrator can create a new ID Controller, so the token they send to the server should contain a field "accountType" and this accountType should be "administrator". We can trust this attribute because it is encoded with our JWT secret. This greatly simplifies controller logic and makes it easier to solely focus on business logic.

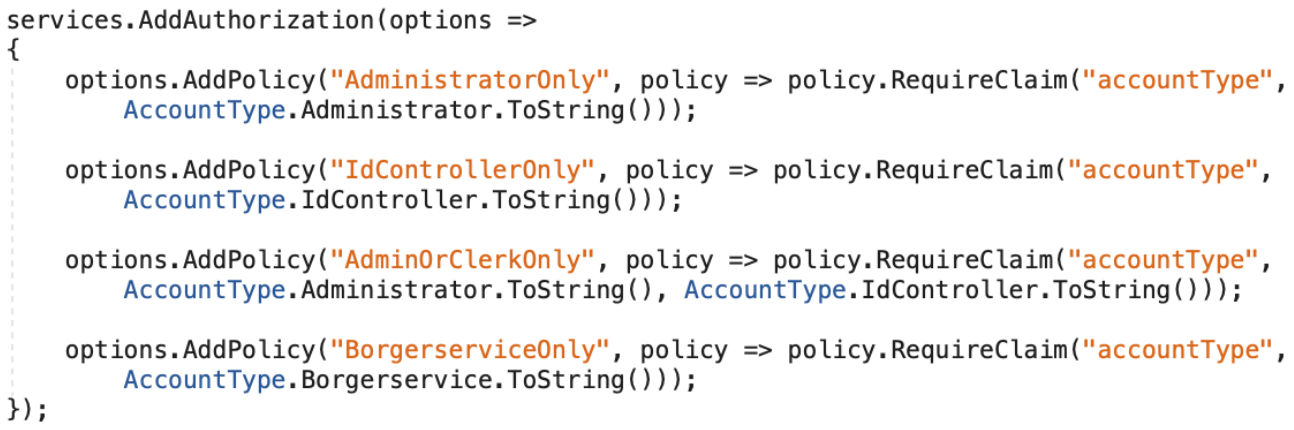
13 Implementation

13.1 Authentication

13.1.1 JWT

Startup.cs adds authentication middleware. This enables us to use the annotiation [Authorize] on our controllers which automatically will respond with a 401 unauthorized if the users' JWT is invalid.

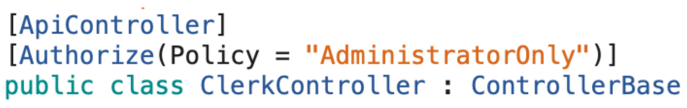
Startup.cs also adds authorization to check claims, which can be seen on Figure 12 below, this enables us to use the annotation [Authorize] with custom policies.



```
services.AddAuthorization(options =>
{
    options.AddPolicy("AdministratorOnly", policy => policy.RequireClaim("accountType",
        AccountType.Administrator.ToString()));

    options.AddPolicy("IdControllerOnly", policy => policy.RequireClaim("accountType",
        AccountType.IdController.ToString()));

    options.AddPolicy("AdminOrClerkOnly", policy => policy.RequireClaim("accountType",
        AccountType.Administrator.ToString(), AccountType.IdController.ToString()));

    options.AddPolicy("BorgerserviceOnly", policy => policy.RequireClaim("accountType",
        AccountType.Borgerservice.ToString()));
});
```

Figure 12: Authorization Policies implementaion

This can be seen in our ClerkController which should only be accessible to admin users. We therefore use the policy "AdministratorOnly" which is the first policy defined Figure 12 above, which automatically checks their JWT claim accountType to be administrator. This is seen on Figure 13 below

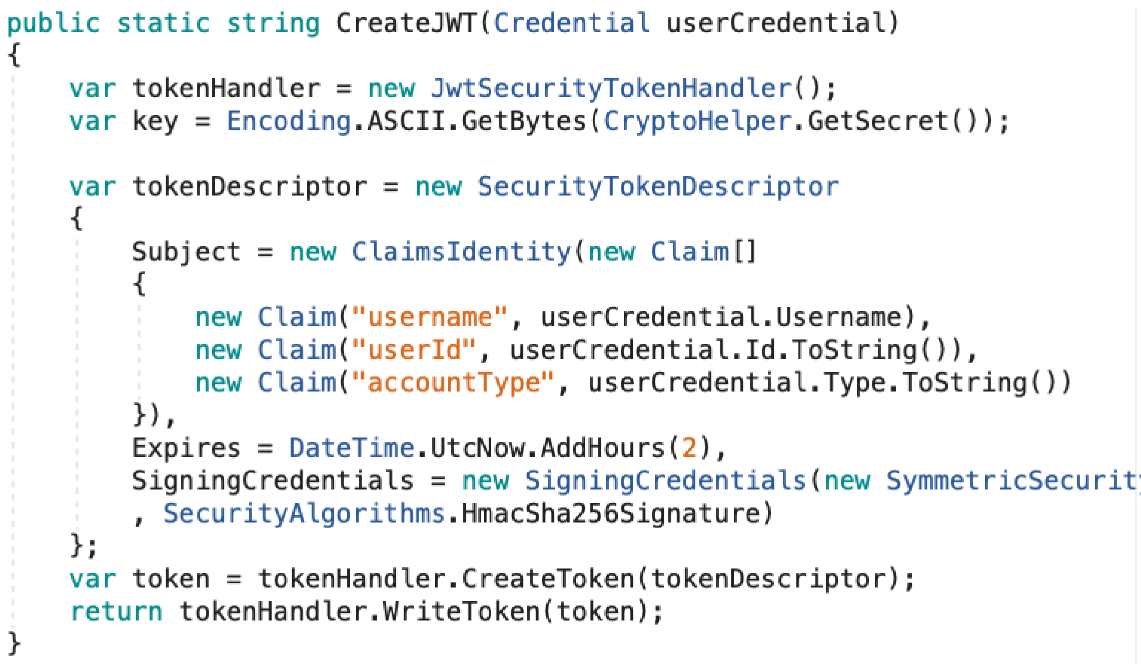


```
[ApiController]
[Authorize(Policy = "AdministratorOnly")]
public class ClerkController : ControllerBase
```

Figure 13: Authorization Annotation Implementation

If a user tries to access endpoints defined in ClerkController and they aren't an administrator, they will recieve a 403 Forbid.

A JWT is made when a user logs in. The endpoint to login is defined in AuthController, which takes a required username and a required password as input and returns a JWT. The JWT is made with a helper class (JwtHelper) which has a static method CreateJWT, which can be seen Figure 14 below.

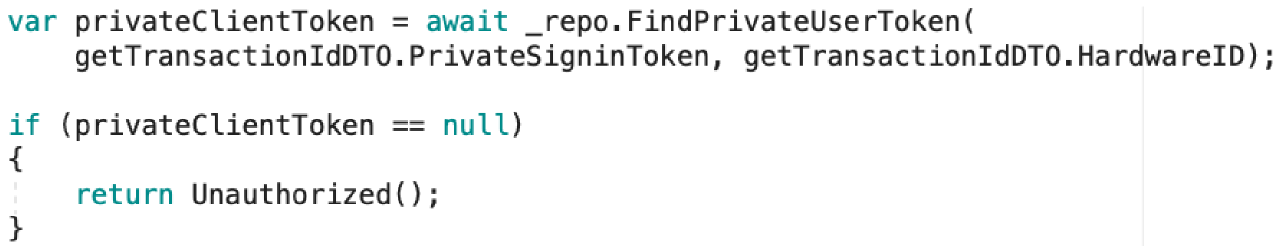


```
public static string CreateJWT(Credential userCredential)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.ASCII.GetBytes(CryptoHelper.GetSecret());

    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim("username", userCredential.Username),
            new Claim("userId", userCredential.Id.ToString()),
            new Claim("accountType", userCredential.Type.ToString())
        }),
        Expires = DateTime.UtcNow.AddHours(2),
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key)
        , SecurityAlgorithms.HmacSha256Signature)
    };
    var token = tokenHandler.CreateToken(tokenDescriptor);
    return tokenHandler.WriteToken(token);
}
```

Figure 14: Helper funciton for creation of Java Web Tokens

13.1.2 Private User Token validation

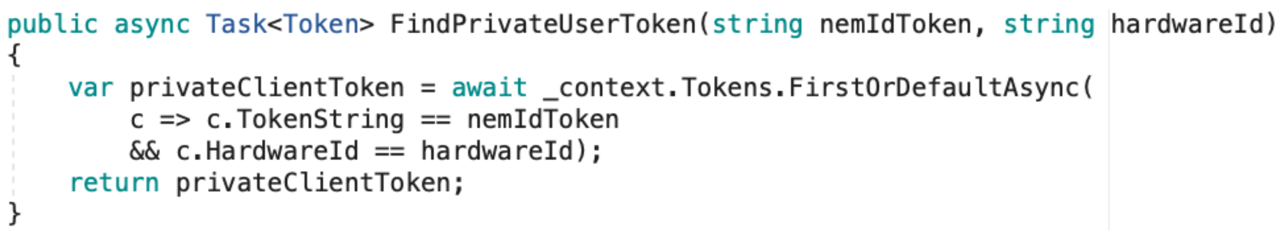
As described in Design chapter, private users aren't validated with JWTs', instead they are required to send NemIdToken and hardware ID when signing up which then is saved in our database and whenever they communicate with our server the NemIdToken and hardware ID is checked up against the database. This can be seen on Figure 15 below.



```
var privateClientToken = await _repo.FindPrivateUserToken(
    getTransactionIdDTO.PrivateSigninToken, getTransactionIdDTO.HardwareID);

if (privateClientToken == null)
{
    return Unauthorized();
}
```

Figure 15: Verification of private sign in token

The repository is using EF Core to check the tables 'tokens', which can be seen Figure 16 below

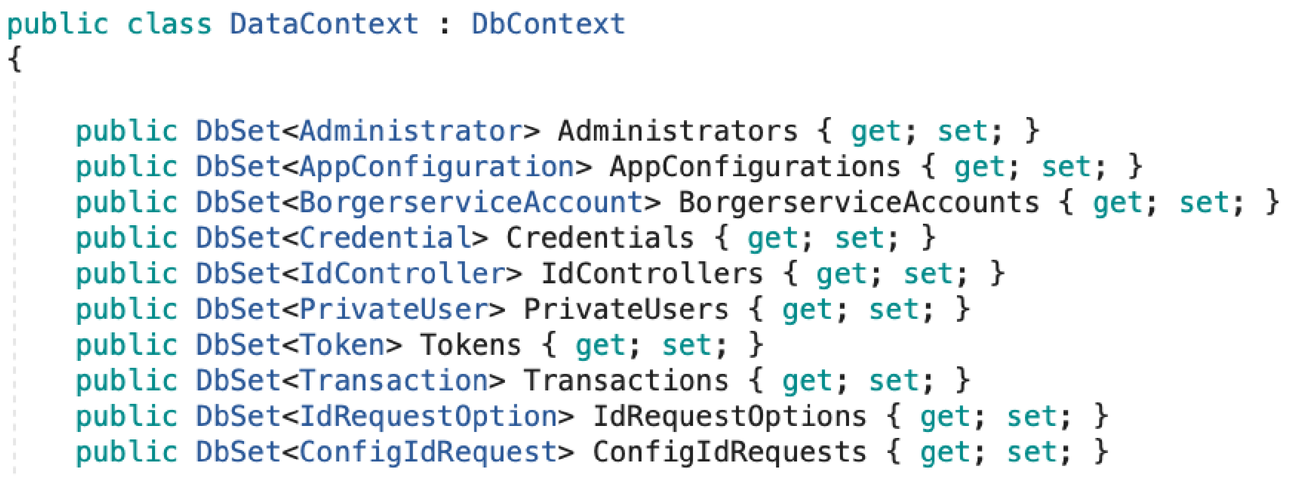


```
public async Task<Token> FindPrivateUserToken(string nemIdToken, string hardwareId)
{
    var privateClientToken = await _context.Tokens.FirstOrDefaultAsync(
        c => c.TokenString == nemIdToken
        && c.HardwareId == hardwareId);
    return privateClientToken;
}
```

Figure 16: Verification of equality of token and id

13.2 Entity Framework Core setup

As described in design chapter, we use Entity Framework Core (EF core), which needs to be configured in a specific way to work properly. The central piece of EF Core is a DbContext (DatabaseContext), which is a class representing database tables and which the developer uses to send commands to the database. This class needs to be extended with the projects specific business logic. This can be seen on Figure 17, where we defined a new class DataContext which inherits from DbContext. This class has DbSet<> properties, where each DbSet is a table in the database. A DbSet property takes an entity class as type.

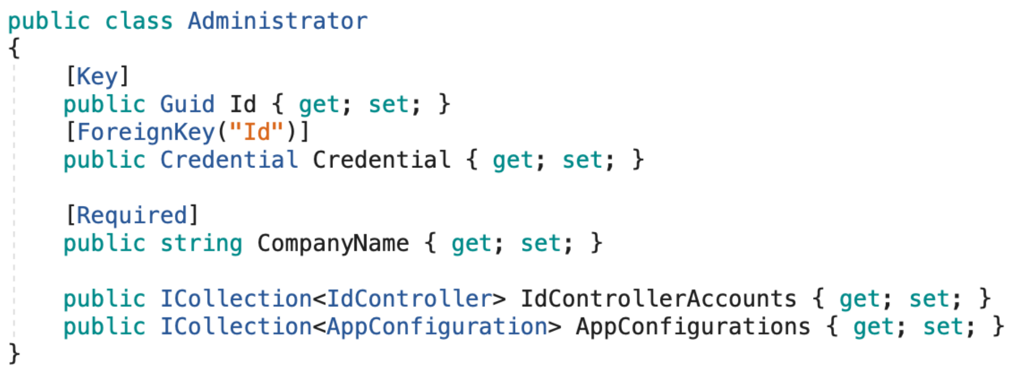


```
public class DataContext : DbContext
{

    public DbSet<Administrator> Administrators { get; set; }
    public DbSet<AppConfiguration> AppConfigurations { get; set; }
    public DbSet<BorgerserviceAccount> BorgerserviceAccounts { get; set; }
    public DbSet<Credential> Credentials { get; set; }
    public DbSet<IdController> IdControllers { get; set; }
    public DbSet<PrivateUser> PrivateUsers { get; set; }
    public DbSet<Token> Tokens { get; set; }
    public DbSet<Transaction> Transactions { get; set; }
    public DbSet<IdRequestOption> IdRequestOptions { get; set; }
    public DbSet<ConfigIdRequest> ConfigIdRequests { get; set; }
```

Figure 17: DB Sets

An example of an entity class can be seen on Figure 18 below, this class represents the table "administrators" in the database. This class has three annotations [Key], [ForeignKey], [Required]. The two annotations key and foreignkey tells EF Core that this table should use the primary key from the credentials table as the primary key in the administrators table.

The entity class also has two collections which is properties used by EF Core to define a many to one relation. These collections will not be found in the database table itself and will be null if not included in a database request to get an administrator. EF Core uses these collections to properly make SQL joins.



```
public class Administrator
{
    [Key]
    public Guid Id { get; set; }
    [ForeignKey("Id")]
    public Credential Credential { get; set; }

    [Required]
    public string CompanyName { get; set; }

    public ICollection<IdController> IdControllerAccounts { get; set; }
    public ICollection<AppConfiguration> AppConfigurations { get; set; }
}
```

Figure 18: The Entity class Administrator

This however is only one piece of the setup for a many to one relation. IdControllers and administrators has a many to one relation. Administrators has a collection of IdControllers and IdControllers needs to contain an AdministratorId Property, which can be seen on Figure 19 below.

```
public class IdController
{
    [Key]
    public Guid Id { get; set; }
    [ForeignKey("Id")]
    public virtual Credential Credential { get; set; }

    [Required]
    public Guid AdministratorId { get; set; }
    public virtual Administrator Administrator { get; set; }

    [Required]
    public Guid AppConfigurationId { get; set; }
    public virtual AppConfiguration AppConfiguration { get; set; }
}
```

Figure 19: Many to on relation example

This setup enables EF Core to include administrators when fetching IdControllers or including appconfigurations when fetching IdControllers, which can be seen on Figure 20 below.

```
await _context.IdControllers
    .Where(ic => ic.AdministratorId == administratorId)
    .Include(ic => ic.AppConfiguration)
    .ToListAsync();
```

Figure 20: Examle of inclusion of other structures

Without inclusion the field AppConfiguration would just be null.

13.3 Controller structure

13.3.1 Repositories

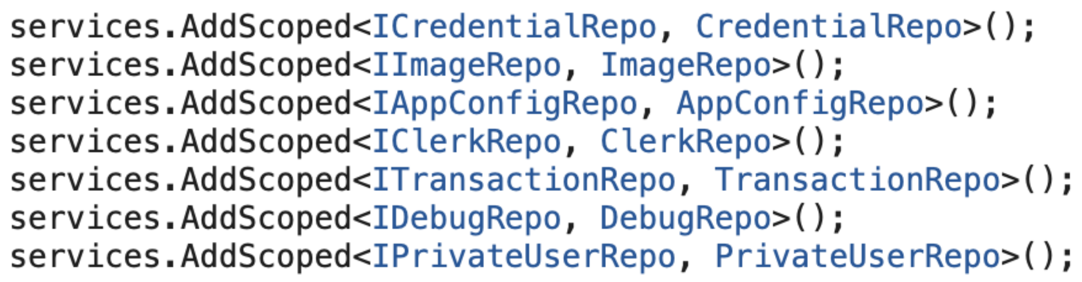
Each controller should have exactly one repository which takes care of all database requests and some business logic. This repository is injected with Dependency injection and each repository has an interface describing the methods the repository should contain. A controller's constructor will declare that it needs a repository which dependency injection will inject; this repository will then be set to a private constant which the controller can use.

In Figure 21 below the interface for an app configuration repository is shown.

```
public interface IAppConfigRepo : IBasicDbOperations
{
    Task<IEnumerable<AppConfiguration>> GetAppConfigurations(Guid administratorId);
    Task<AppConfiguration> GetAppConfiguration(Guid administratorId, Guid appconfigId);
    Task<AppConfiguration> GetAppConfigForClerk(Guid idControllerId);
    Task<AppConfiguration> Delete(Guid appconfigId, Guid administratorId);
    Task<IEnumerable<IdRequestOption>> GetIdRequestOptions();
}
```

Figure 21: App configuration repository interface

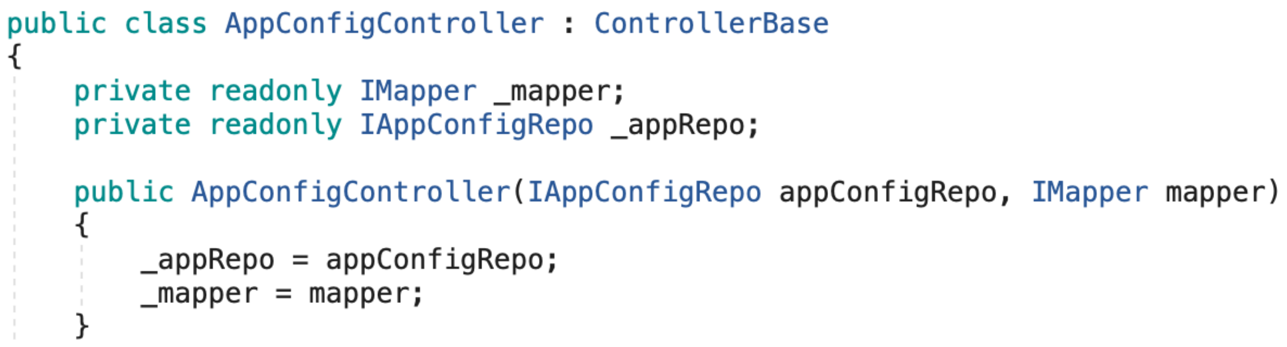
Which a class called AppConfigRepo.cs implements. In startup.cs it is declared that this concrete implementation can be used when IAppConfigRepo interface is needed. All the repositories can be seen declared Figure 22.



```
services.AddScoped<ICredentialRepo, CredentialRepo>();
services.AddScoped<IImageRepo, ImageRepo>();
services.AddScoped<IAppConfigRepo, AppConfigRepo>();
services.AddScoped<IClerkRepo, ClerkRepo>();
services.AddScoped<ITransactionRepo, TransactionRepo>();
services.AddScoped<IDebugRepo, DebugRepo>();
services.AddScoped<IPrivateUserRepo, PrivateUserRepo>();
```

Figure 22: Declared Repositories

AppConfigController.cs constructor declares that it requires an IAppConfigRepo, which can be seen on Figure 23.



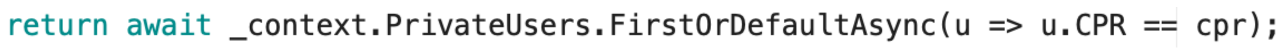
```
public class AppConfigController : ControllerBase
{
    private readonly IMapper _mapper;
    private readonly IAppConfigRepo _appRepo;

    public AppConfigController(IAppConfigRepo appConfigRepo, IMapper mapper)
    {
        _appRepo = appConfigRepo;
        _mapper = mapper;
    }
```

Figure 23: Declaration of requirement

13.3.2 Async

ASP.NET is multithreaded by default so in a simple case with two users it makes no difference if we use async controllers' endpoints, however with a lot of users it blocks the thread when making blocking calls such as database calls or waiting for user input. When using async methods it's possible to await a blocking call, which waits for the database to return on a background thread so the main thread can continue serving other requests.

In C# a Task is something which is awaitable, so when the repositories make database calls, they return a Task, which the controller can then await and when the database call returns the requested object the thread will go back to where it stood waiting which completes the task. This can be seen on Figure 24. While EF Core communicates with the database and finds a private user with the correct cpr the thread can use resources on other requests



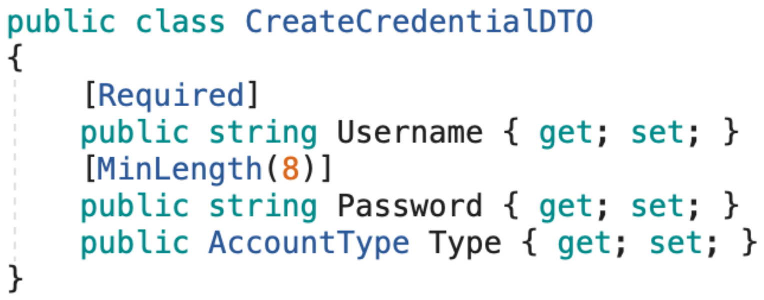
```
return await _context.PrivateUsers.FirstOrDefaultAsync(u => u.CPR == cpr);
```

Figure 24: Async operation

13.3.3 DTOs'

A DTO stands for Data Transfer Object and is heavily used in this project in combination with Automapper. When creating a new Credential, the user has to enter the fields described in CreateCredentialDTO, the idea is that the classes representing object in the database (entities) should not be filled with business logic, like password requirements as the password in the database is just a hashed password. Such business logic will then reside in a DTO instead. Other things include proper string representation such as a Tostring method or JSON annotation, anything that isn't related to the database tables should not be in the entity classes.

CreateCrendetialDTO can be seen in figure 25 below, Username and Password is required, and the password should minimum be 8 characters long. Here it's also possible to check for password strength such as minimum one small and large character, one special character etc. These details should never be defined on an entity class, as it's pure business logic. DTOs' are great for this, especially with the user of AutoMapper as well where it's easy to copy values from a CreateCredentialDTO to a Credential object which can be saved to the database.



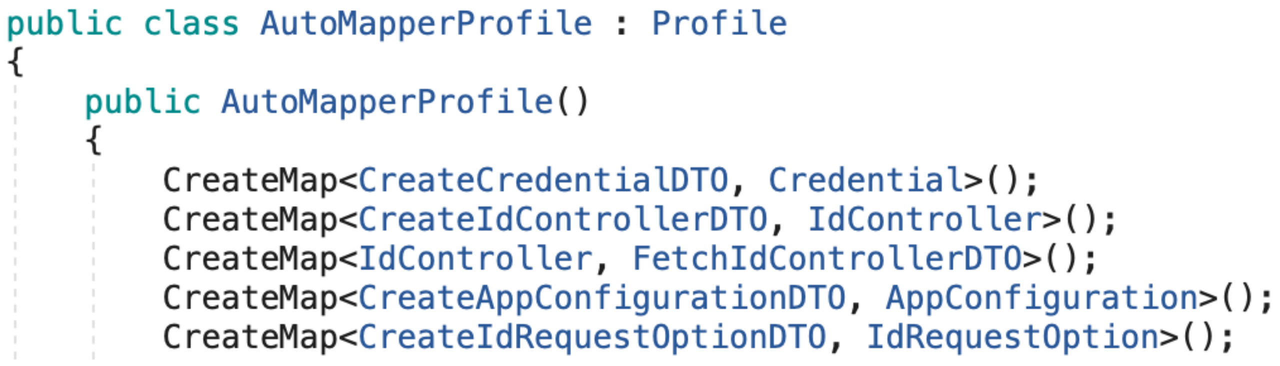
```
public class CreateCredentialDTO
{
    [Required]
    public string Username { get; set; }
    [MinLength(8)]
    public string Password { get; set; }
    public AccountType Type { get; set; }
}
```

Figure 25, CreateCredentialDTO

However, some business logic has creeped into our entity classes without a good reason, it is however very minor fixes, and doesn't really have any impact on this project.

13.3.4 Automapper

Automapper is easy to use and saves a lot of time and code. In Startup.cs Automapper is configured so that controllers can get an IMapper injected, this mapper needs to know which maps it can do. An AutoMapperProfile class is used which declares these maps.



```
public class AutoMapperProfile : Profile
{
    public AutoMapperProfile()
    {
        CreateMap<CreateCredentialDTO, Credential>();
        CreateMap<CreateIdControllerDTO, IdController>();
        CreateMap<IdController, FetchIdControllerDTO>();
        CreateMap<CreateAppConfigurationDTO, AppConfiguration>();
        CreateMap<CreateIdRequestOptionDTO, IdRequestOption>();
```

Figure 26: AutoMapperProfile

As can be seen on Figure 26 above. We declare that Automapper should be able to take a CreateCredentialDTO object and map values from that object to a new Credential object as described in DTOs' section above.

For automapper to map values between two different types, the properties must match, so for automapper to map Username and Password from a CreateCredentialDTO object to a Credential object. Credential must have the same properties Username and Password with the same types and property names.

14 Security

Seeing as our system uses and stores sensitive data, security is of the highest priority. This means we have followed the Design Checklist for Security from (Bass, 2013), as well as used the OWASP guides[6] to secure the applications in our system. We don't store any of the personal data, other than the user-id linked to their social security numbers, in our database.

Other than the obvious security measure of using TLS encryption for all traffic, we have also done other things. One such thing is that all passwords have been hashed and salted with a per-user salt. We have gone further and used variable salts enabling us to secure our user-credentials, should we be hacked, against rainbow-tables[7]. We never send passwords in clear text, even inside TLS. This means that even if all our systems to be compromised, we would never have any user's password. We have further protected our applications and users by using a real domain and getting a real certificate from a valid CA[8] so that none of our applications uses self-signed certificates or ignores certificates altogether.

One of our early decisions were to avoid having any more sensitive data that the CPR-number, and then fetching all necessary data live from the CPR-register when needed. This means that if a compromise were to happen, only the CPR would be leaked, not any other personal data. It has the additional benefit of avoiding outdated data being used because of delayed updates. We do however have access credentials to the CPR system on our backend, meaning that if our system was to experience a total system-takeover before we would notice, they would have access to the CPR register to the same degree as the system, before our systems access keys could be invalidated. This is a major risk, but no more than any similar system that depends on the CPR register for sensitive information. In fact, because we only have the information available in the CPR register with access keys, there would be no way of doing a database dump, and any brute force database dump would be detected. Once detected all access to the CPR register could be removed instantly leaving the attackers with only CPR numbers and no other data.

We could mitigate some risks by encrypting the CPR numbers in our database, such that only the backend would have the key. That means that should our database be compromised, then no information is available without full control of the backend. This would remove some of the danger of an SQL-injection attack.

Right now, our system does not record failed login-attempts or audit-trails. This should be added before the system would be usable in a real-world scenario as it would make it possible to determine if an attack is occurring and whether an employee is misusing the system. Our data is system is separated into containers for several reasons, one of them being that should an attacker gain control of the host machine through an unknown exploit in the webserver, then there is no access to the other parts of the system, limiting exposure. Right now, we have no backup-system, to recover and restore from attacks. Such a feature should also be implemented.

As our system primarily used well-known implementations of secure algorithms and secure frameworks, we have trust in the fact that there should be very few if any exploits into our system. To

[6] OWASP is a foundation creating guidelines for securing applications, https://www.owasp.org
[7] Lists of precalculated hashed of various bruteforce patterns and password lists
[8] Certificate authority

continuously believe that, we must maintain up-to-date software, so a feature for that should be implemented.

15 Test

We tested our software in a few different ways over the course of development. Build testing was initially the only form of testing done, but we went over to manual testing as we began to have more methods, interfaces and endpoints. We began cursing ourselves as we approached the end of development for the backend, as the amount of work required to test a single endpoint had increased greatly, but we decided not to start writing unit tests as the development was practically over for the backend and therefor did not need to be altered, and as there were no alterations in sight due to the scope of the project, we decided that the time needed for potential future tests were less than the amount of time needed for writing complete unit/integration tests. The manual tests of the backend were performed by calling the backend with either the frontend, the apps or an application, Postman. We also made the testing easier with functionality build into Postman[9], such as saving our json web tokens or saving things like transaction id's. This saving and transferral of information between calls made the testing far easier, and significantly lowered the test time.

16 Discussion

In this section, the results of the project will be discussed by reflecting on our initial requirements to DigID, both functional and non-functional. How the main problem presented in the introduction is solved by the product will also be discussed.

16.1 Functional requirements

Throughout this project, we have succeeded in creating different pieces of software to realize DigID. However, we have not met our goal of also making an internal client and an eventual API for third parties to integrate. These components were not as important than others and had lower priorities. The internal client is not necessary when demonstrating the system DigID. The purpose of this component was to make the internal stakeholders, like us, able to manage which companies exist in the system. This is far from the core purpose of this system and would therefore only contribute slightly to this project.

The API for third party developers to integrate in their own systems could have given this project a lot of potential. This would make our solution more flexible and usable in almost endless situations. However, this component had the lowest priority because it required a great deal of the system to be finished beforehand. If it had been a real-life project this would have been taken into consideration a lot more, and likely would have had a huge impact on the software architecture.

The parts of DigID that we developed turned out very well. The private user app and the company app are both useable and able to communicate with a backend server to handle transactions. They both meet all their requirements. The same applies for the Borgerservice client and the Administrator client apart from altering ID controller accounts and app configurations. Generally, the system is functional and somewhat complete.

[9] Postman allows for test environments, values can be saved and read from variable in these environments.

16.2 Non-functional requirements

While analysing the problem and system, the group defined a few non-functional requirements that DigID should meet. These requirements mostly regarded the system's security and performance. Requirements such as these are difficult to measure, but it is possible to reflect on them. Non-functional requirement F02 states that the system should store data in a way that is unreadable by unauthorized people, for example by encrypting. This requirement is partly met by DigID. In the database, the passwords for the administrators, clerks, and Borgerservice employees are hashed and salted. If a malicious user should get access to the database, he would not be able to read any passwords or access people's accounts. Though, the rest of the fields are not altered to obscure the data. This is not ideal because the database stores the CPR numbers of the private users. Here, the group would have liked to implement AES encryption to make them more difficult for malicious users to read.

Requirement F03 is very similar. It says that the system must make it difficult for malicious users to hack others. Our system ensures that only authorized users have access to their own functionality with the use of JWTs. This means that only an ID controller can request personal data, and so on. Even if an ID controller is intentionally trying to abuse the system by requesting more personal data than necessary, the private user can view and decline the request before it has gone through. Also, this system is made trustworthy by using server-side verification. Every piece of data must go through the backend server, which cannot be easily tampered with like the clients. This means that it is not possible for private users to hack the system to, for example, make them appear younger or older than they actually are. DigID could have been made even more resistant to attacks by recording failed login attempts and audit-trails. As of right now, it would be difficult to notice any attacks happening on the system.

Requirements F00 and F01 regards DigID's performance. It is especially difficult to say whether these requirements are met as it depends on many things. Regarding requirement F01, which states that the system should not take long to handle transactions between a company and a user. Because the system is distributed, there will always be a slight delay when the clients communicate with the server. It is especially bad with this system as a transaction requires the server to communicate with several different applications over HTTP; the private user app, the company app, and the CPR register. However, even with these conditions, the transactions are handled acceptably quickly. Though currently, the server runs on the same machine as NemID and the CPR register, but in a real-life scenario it would use the official NemID and CPR services which would likely make the transactions take a little longer time.

Requirement F00 states that the system should not take long to open and be ready for a transaction to happen. This requirement is seen from a user perspective and affects how quickly the private users can get a transaction started. Because the private user app is able to be unlocked by a user defined pin code, the waiting is shortened a lot. The unlock functionality requires no communication with the backend or any other services and does not have to wait for HTTP requests. This results in quickly being able to get the app up and running. The same could not have been achieved if users had to sign in with NemID each time.

Requirement F05 regards the system´s availability and says that the servers must be able to handle 5% of users at the same time. To help make DigID able to handle several consecutive users, we have used async methods. This allows the main thread to continue serving requests while other threads handle the blocking calls. However, the exact amount of people this can handle is not possible to know.

The rest of the non-functional requirements revolves around miscellaneous qualities, such as the apps need to work on Android, be energy efficient, and work with a slow internet connection. These seem to be met; however, it is difficult to say whether the apps are energy efficient. The DigID backend was also required to have a modular design to allow for easy 3rd party integration. However, as described earlier in the discussion, there was little focus on this during development. F06 states the users must not be able to change any of their information with the application. This requirement is met due to the system's distribution. Personal data is never stored on the clients but always requested from the CPR register. As to whether the users need to have their pictures taken at an authorized place, it is not possible for themselves to choose their own pictures. This is handled by authorized Borgerservice employees through the Borgerservice web client. Currently, there exists no terms of service for DigID. This would be required in the real world, as DigID handles personal data. We chose not to focus on this requirement as it does not help demonstrate the system's core functionality.

16.3 Solving the problem

In the introduction, the following main problem to this project was presented:

> *How could one streamline identification, such that users do not disclose unnecessary and sensitive information and enable users to carry identification on their phone in a secure enough way for it to be recognized as legitimate?*

The first part of this question regards changing the identification process to something that only shows information that is relevant for the particular situation. The solution the group has created, has shown that this is possible by letting the id controller choose exactly what data is needed. This also removes the need for customers to carry their physical identification around everywhere. They still need to bring a phone, however, but this is very common today.

The question also states that it needs to be secure enough to be recognized as legitimate. Several aspects of DigID participate in making the system secure. The use of JWTs ensures that only authorized users gain access to the system and that they gain access to functionality relevant to only them. Also, by giving the private users the ability to view what information is requested by the id controller before the transaction happens, enables the private users to control exactly what personal information they want to disclose. Overall, using server-side verification makes the solution more trustworthy to use and more difficult to hack.

To conclude, the group has succeeded in constructing a solution to the main problem, throughout this project. The distributed system DigID makes the process of verifying yourself more reliable, safe and trustworthy than ever before.

17 Market analysis

In this section it will be discussed what opportunities the product has in the market. First, DigID will be compared to other already existing solutions based on the result from Other solutions. It will also be discussed well DigID will fare in an international market and how its international capabilities could be improved in the future.

Compared to the other solutions on the market, the product of this project has several advantages. Unlike the conventional physical identification, this system only reveals the personal data that is required in the particular transaction. In contrast to SmartID, DigID is more trustworthy. This is accomplished through the server-side verification in addition to not letting the users choose their own photos.

As of right now, DigID is not suited for use outside Denmark for several reasons. The DigID Private Client app uses NemID for registration of new users. Because NemID only exists in Denmark, users from other countries would not be able to use our system. The DigID private client would need different registration systems for each country. In this project, we imagine that DigID is developed by the state because it handles personal data from the CPR register and essentially stretches the laws of identification. Therefore, the registration systems used need to be approved by the state and preferably in use already in the given country. This complicates the process of internationalizing the product even more as it would be almost impossible to develop a standardized registration system that is approved and works across every country. In the Scandinavian countries, the problem should be less significant. Both NemID and the equivalent systems in the other Scandinavian countries are developed by Nets (Nets, 2019). This should make it easier to replace the already existing NemID with a system that works in these countries. Currently, this will be challenging in the rest of the world. However, a new European standard is on the way that could replace NemID (Knudsen, 2019). By implementing this in DigID, the entirety of Europe could gain access to our product.

Currently in the ID-controller app, you choose 18+ if you for example want to sell cigarettes in Denmark. However, these age restrictions will likely be different in other countries. A way to make it more flexible is to make the app automatically decide the correct age restriction based on where it is located. This would allow the clerk to choose a picture of a cigarette and let the app do the rest. To make this possible, the system would need a collection of age restrictions for each country. For example, if you want to buy cigarettes in Belgium, the age limit is 16+ (Demetra, 2019). This would then be a part of Belgium's country configuration.

To make the clients suitable for other cultures, they will need to be compatible with multiple languages. This is currently not the case with the entirety of DigID, as only the mobile apps support more than one language. In the prototype, the default language is English. However, if it detects that the user is from Denmark, the language will change. This is a built-in feature in Android that detects what language your phone is set to (Support.google.com, 2019). Therefore, the foundation for a multi-language application is already laid, but the languages just need to be added as the system is implemented in more countries. However, a similar system also needs to be added to the Administrator and Borgersevice web pages before DigID is fully compatible with multiple languages.

Currently, DigID will not work outside of Denmark. The differences in laws and regulations result in a different app configuration needed for every country. It will be difficult to make DigID suitable for an international market. However, future European standardized solutions might increase its opportunities in the European market.

18 Product Evaluation

This evaluation will not take design choices such as colours, button placements etc. into account, as the design is subject to change a lot – it is only a prototype. It is only the core of the product, which is evaluated, the premise, the benefits and the potential disadvantages.

18.1 Advantages

Like any other digitalization of a physical product, there are groups who aren't skilled in using IT which will be dissatisfied if a physical product is completely replaced by a digital product. Products like ours can potentially have consequences for these citizens if it replaces physical cards sometime in the future. But the system will most likely never completely replace physical cards completely as phones are unreliable compared to a physical card. A physical card can't unexpectedly shut down or rapidly lose power. Therefor most people using this system will most likely continue to bear their physical cards as well, and the two solutions will most likely exist as a backup to the other. If a user has forgotten their phone, they can use their physical card, and if a user has forgotten their wallet, they can use their phone.

This product feels like the right step towards digitalization, it feels natural to start replacing physical cards. This can be seen as the Danish government has taken steps to digitize driver's license (Finansministeriet, 2018). This is however different from country to country. It makes sense as most people always have their smartphone near them.

If users trust their government, it can be convenient to use a system developed by their government to handle their identification. The system will enable the users to be more private, they won't have to expose extra information on the card, if their card holds their address and name, and all they need to show is their name. It can also have the opposite effect; some people may never trust their government to also handle their identification.

It can be convenient to use an app on one's phone rather than having to find one's wallet. Especially in Denmark as most postal companies send a text with a code which is used by the clerk to identify the correct package to retrieve, therefor the mobile is already in hand and ready to be scanned for identification also.

This system would also benefit greatly for people traveling if it could be used in multiple countries.

18.2 Downsides / Future implementations

A huge potential downside is availability in the form of acceptance. If only a third or half of all stores use and accept identification with this system in a given country, it won't be used by the citizens. In order for it to work properly and be used, every store has to use and accept it. This can only be achieved if the government makes a new law or removes physical cards, the most likely is that they make a law requiring stores to accept this system as a valid identification system which can be problematic.

A lot of mixture of Danish and English is currently a big issue in all the applications. This would be a focus in future implementation sprints. Translate all Danish words to English, so the default language is English.

Another huge downside and future implementation fix for users is business logic which is only relevant in Denmark. Danish terms such as "borgerservice" which is a service that represents the Danish government and enables citizens to resolve issues related to their citizenship, needs to be removed, so the product doesn't cater towards a specific country.

A potential annoyance is when the user opens the app to generate a QR code, gets the QR code scanned and get asked if they want to share the information the controller is asking for. In most cases the user will always accept the request, as the controller in most cases is not trying to steal any information. As an example, if a user frequently visits a postal office to retrieve packages and they have to accept each time, it can get annoying when they trust the postal office. It could therefore be a potential future implementation to enable users to set an option to always trust that specific postal office to freely request their address or name or any other piece of information. This would allow the user and postal office clerk to resolve the interaction quicker.

The company client can use their camera to scan, however they can't zoom or click to focus, this would be a feature worth implementing for user experience.

If it costs a store owner money to use this identification system, it can be a tough sell as most of the benefits for the store owner is easier age identification as clerks won't need to calculate if a person is old enough to buy alcohol based on their birthday. It can of course be a good investment if users are loving the system and therefor prefer stores which implement it over other stores.

19 Process Evaluation

In this section, we will discuss the process used to structure the development in this project, and whether it has worked as intended or not.

When developing the different components of DigID, the group split into pairs. One pair might have overseen the development of the Private Client while another developed the Company Client. Within each pair, the people worked together on the particular component through pair programming. This method proved to be very efficient, as the group was able to get a lot of work done simultaneously. However, we did encounter some problems with this process. The pair programming was not as efficient as we would like. It often ended up with one member of the pair programming while the other just watched passively. Pair programming works well when people need to discuss how things should function, but not when things need to get done. Therefore, it often made tasks take much longer than originally intended. To improve the pair programming, the pairs may discuss the work beforehand and then program individually from there.

Another problem with this method of dividing labour was that some of the groups depended on each other's work. For example, to make the Private Client work, the Company Client needed to be done and the other way around. This introduced some difficulties that needed to be solved across the pairs. Next time, we need to be more careful with how the labour is divided between the pairs.

Sometimes, the group members did not agree when each component was finished. Few pieces of the software had not been tested properly by the developing pair and therefore posed errors when used. This led to other group members spending unnecessary time trying to understand the code and to test solve the problems. This could have been prevented if the group had agreed on a "Definition of done" beforehand. Next time, it would also be possible to run a Test-Driven Development. Here, tests are the first thing to be written, and afterwards code is written to make the tests succeed. Then the team will always have tests available to ensure the code works as intended even after new changes have been made to it.

From the beginning, the group intended to use Scrum. We have been somewhat successful with this. We managed to create sprints of approximately two weeks. In the beginning of each sprint, the group created a backlog of all the tasks that needed to be done by the end of the sprit. This worked very well. However, the group was not good at practising all the Scrum ceremonies. This led to very little coordination across the pairs, and often you did not know what the other team was working on. Sprint review was only practised sometimes which made it even more difficult to know what the other pairs had been working on. However, the group did code review each other´s work from time to time, and this contributed to understanding parts of the system you did not personally work on.

Overall, the project ended up with a product that worked as intended. However, the process of making it could have been better. By utilising Scrum's ceremonies more and increasing the communication between the developers, it might be possible to do even better next time.

20 Conclusion

By creating a digital solution that can safely verify the social security number, it is possible to only share the necessary information whenever you are at a kiosk and need to show id. The system presents the required information about a given person, rather than exposing a personal id card with unnecessary information for the given situation.

In order to be able to trust an application on the user's phone, the business logic will be handled on a server, in an encapsulated environment that has a limited interface. During the project this was achieved by implementing a distributed system architecture, using client-server and microservices.

If the system should approach international markets, there are some limitations that needs to be dealt with. First of all, of the language barrier. The android apps have already implemented multiple languages, but the web application should do so as well. Apart from that, the AppConfigurations are currently bound to Danish laws. This should be made more flexible. And lastly, the system uses the Danish service NemId in order to verify the user – A similar service must be available in any international market in which the system should function.

The project is built on a number of assumptions. It merely shows that it is technically possible to build a secure solution that solves the problem. For it to be achieved in a real world, it's safe to assume that it would be a much lengthier process to implement a new valid Danish social security number system like this, and it would most likely have to live up to some standards that are not considered within the limits of this project.

20.1 Perspective

The prototype that has been built lives up to most of the initial requirements. Most noticeably there wasn't build an API interface for 3rd party developers, but besides that, only minor implementation-level requirements are pending due to prioritization. The 3rd party API feature would open up for uses cases such as 3rd party door access control integration i.e. on larger workplaces.

Should the project continue development, with the goal of being a production-ready application in the real world, there would be added some additional requirements. Amongst those requirements, one of the highest priorities would be network security and encryption of data. Another requirement would be to get access to and integrate the actual NemID and CPR services.

Out of context of the DigID solution, the way the system relies on server-side validation could be used in other types of applications as well. In this system, the server is used to validate authentication of a user, but it could span much broader than that. Imagine applications where multiple clients try to provide information to the server, and the server must decide the truth and respond back. This could be use in game development where i.e. a soccer game server must determine if the ball in an online game crossed the goal line or not. Or in stock trading, where the server could determine if the user successfully sold a stock before its value plummeted. The key is to have a single source of truth that is responsible for certain business logic.

21 References

Bass, L. (2013). *Software Architecture in Practice, Third Edition.* Pearson Education, Inc.

Bergeon, D. (2018, maj 28). *Taenk.dk*. Retrieved from https://taenk.dk/test-og-forbrugerliv/digitale-tjenester/beskyt-dit-cpr-nummer/hvem-maa-bede-om-dit-cpr-nummer

Demetra. (2019, November 22). Retrieved from https://www.demetra.dk/aldersgraenser-for-koerekort-tobak-og-alkohol-i-europa-og-usa/

Det Centrale Personregister. (2019, 11 22). *CPR Services*. Retrieved from cpr.dk: https://cpr.dk/kunder/private-virksomheder/cpr-services/

Finansministeriet. (2018, 09 23). *www.fm.dk*. Retrieved from Finansministeriet: https://www.fm.dk/nyheder/pressemeddelelser/2018/09/digitalt-koerekort

JWT.io. (n.d.). *JWT*. Retrieved from https://jwt.io: https://jwt.io/introduction/

Kharenko, A. (2015, 10 9). *Monolithic vs. Microservices Architecture*. Retrieved from https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

Knudsen, A. B. (2019, 02 06). *Nu er vejen banet for fælles europæisk NemID – ny standard sikrer krav til kvaliteten af elektroniske signaturer*. Retrieved from Dansk Standard: https://www.ds.dk/da/nyhedsarkiv/2019/2/nu-er-vejen-banet-for-faelles-europaeisk-nemid

Nets. (2019, 11 22). *Digital identities in the Nordic countries*. Retrieved from Nets.eu: https://www.nets.eu/solutions/digitisation-services/identification

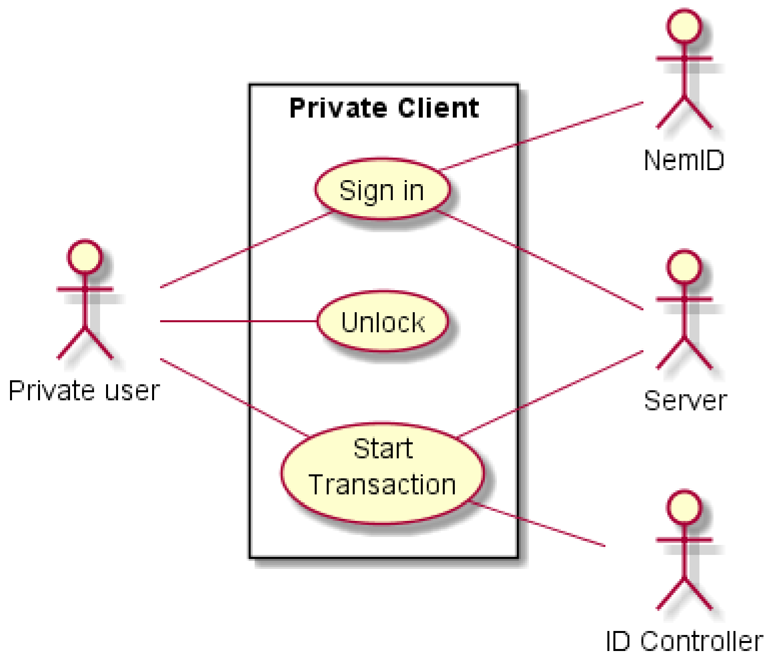
Support.google.com. (2019, November 22). Retrieved from https://support.google.com/googleplay/android-developer/answer/3125566?hl=en

Visbøll, S. M. (2017). Test af digitalt id-kort: Man kan nemt snyde med det, og det har startproblemer. *Politiken*.

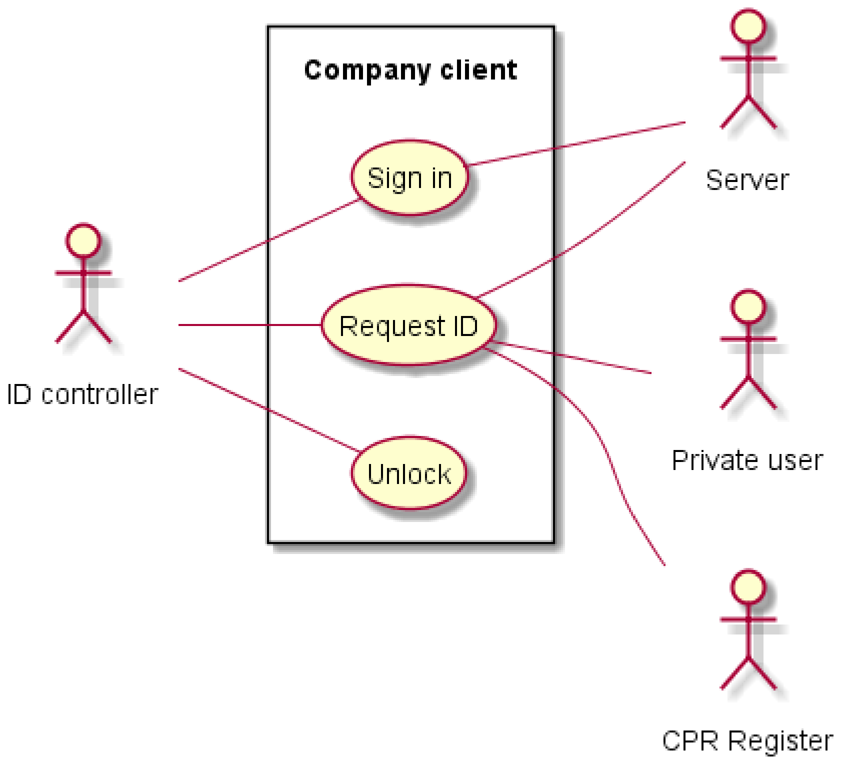
22 Appendix

22.1 Appendix 1 - Use case diagrams

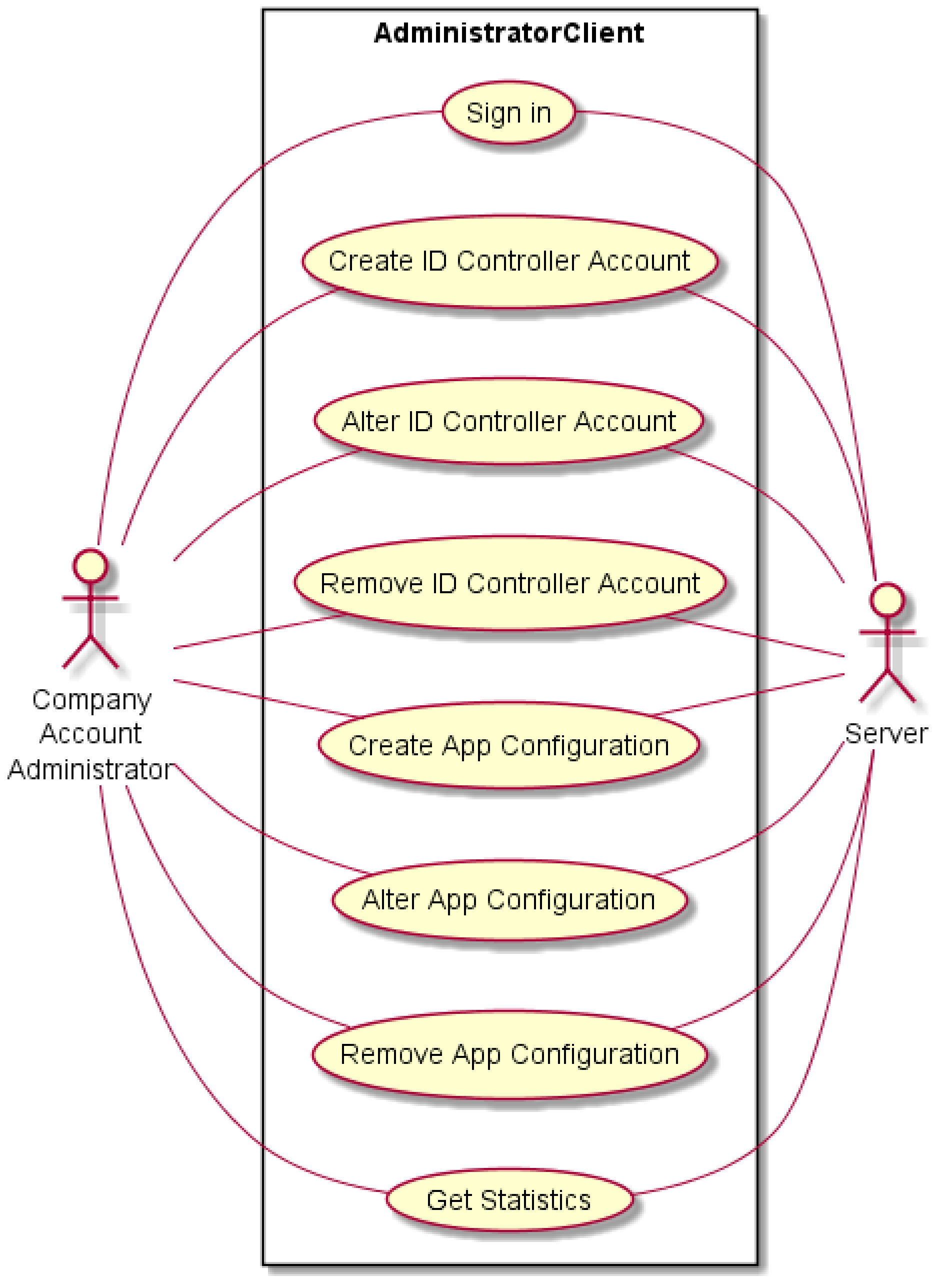
22.1.1 Private client use cases



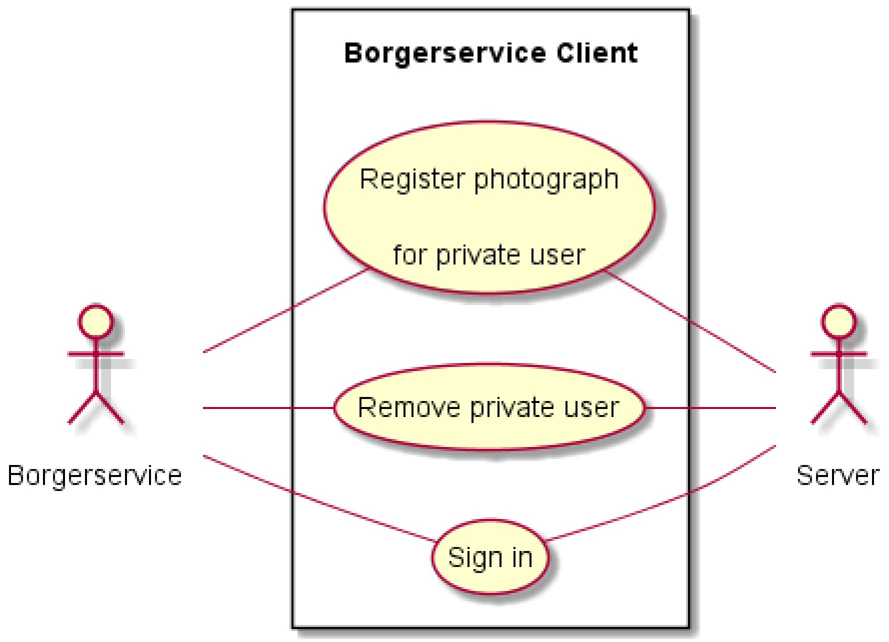
22.1.2 Company client use cases



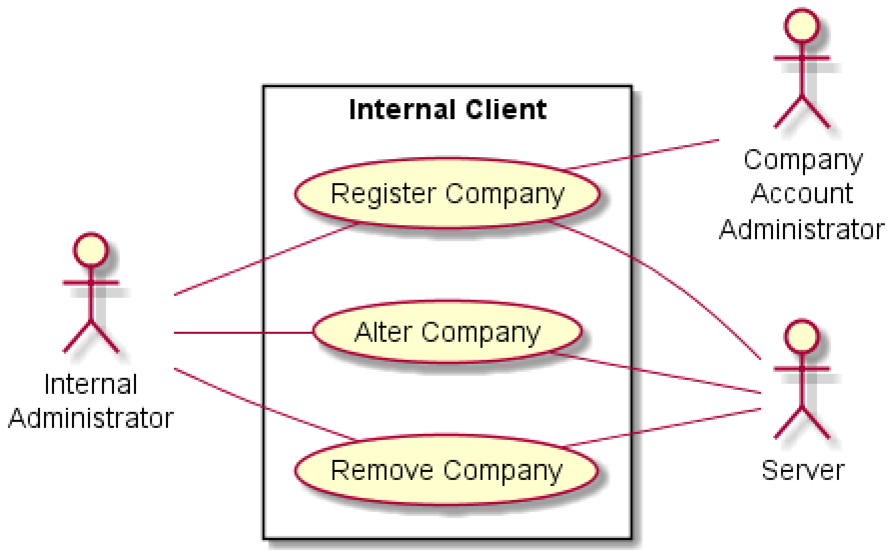
22.1.3 Administrator client use cases



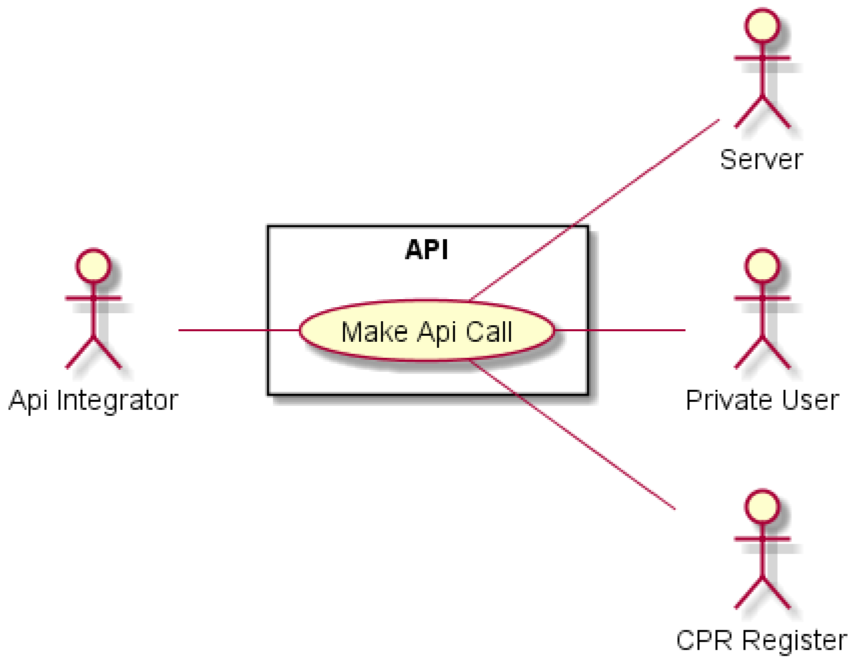
22.1.4 Borgerservice client use cases



22.1.5 Internal client use cases



22.1.6 API use cases



22.2 Appendix 2 – Project foundation

DigID Project Foundation

Group number:	12	
Group advisor:	Henrik Hoffmann	*hhof@mmmi.sdu.dk*
Group members:	Frederik Alexander Hounsvad	*frhou18@student.sdu.dk*
	Jakob Gamborg Jørgensen	*jajoe18@student.sdu.dk*
	Thomas Steenfeldt Laursen	*tlaur18@student.sdu.dk*
	Oliver Lind Nordestgaard	*olnor18@student.sdu.dk*
	Nicolai Christian Gram Rasmussen	*nrasm11@student.sdu.dk*
	Oliver Marco van Komen	*olvan18@student.sdu.dk*

Background/motivation

Buying alcohol at a bar or retrieving a package requires identification to let the clerk know that the customer is old enough or the correct recipient of a package. The only option here is to show either a driver's license or a passport; both of which include several pieces of personal information, such as the entire social security number. The clerk behind the counter gets to see all of it, and the customer must trust him with keeping it safe and not abusing it. In some countries, personal information falling into the wrong hands can be dangerous. For example, in the United States, many banks require only the social security number to set up bank accounts, credit cards, and loans[10]. This method of verifying someone's age or identity should be a controlled process where the clerk only gets to know what he needs to know.
When checking if the customer is old enough, the clerk is prone to making mistakes. Calculating people's age based on their birth date can be difficult. By eliminating the need for the clerk to see the actual identification, the risk of misjudgment will diminish. By removing the need for the customer to bring an id altogether will make life easier for everyone. Having to remember your id everywhere you go can be inconvenient.

Stakeholders

The following analysis groups stakeholders into 3 categories.
Internal stakeholders, which consists of people who are working on the project, and have insights not shared with the other stakeholders.

It should be surprise that the project group and advisors are internal stakeholders, but also the Danish government falls into this category, as the project assumes that they have agreed to form a partnership around the product.

Primary stakeholders, who will directly engage with the product, and be partners or customers. This group of stakeholders are vital for the success of the project. Supermarkets for example will both use our product in their daily routine and will be potential customers for integrating our solution into their own software.

The primary stakeholders map directly to our use case actors (section 6 – Initial Requirement Analysis). Borgerservice as a stakeholder corresponds to Borgerservice as an actor, but a group of stakeholders like Kiosks, Bars, Postal Service, Casinos, and Supermarkets map to the actors Administrator (managers) and ID Controller (workers on the floor).
External stakeholders, which in some degree will be affected by the project. These stakeholders are not directly associated with the project but can be taken into consideration when designing the product.
The list of external stakeholders includes authorities like the police department, and other people who might need to verify one's identity. There are also external users to take into consideration like tourists who does not have a Danish citizenship.

[10] https://www.lifelock.com/learn-identity-theft-resources-kinds-of-id-theft-using-social-security-number.html

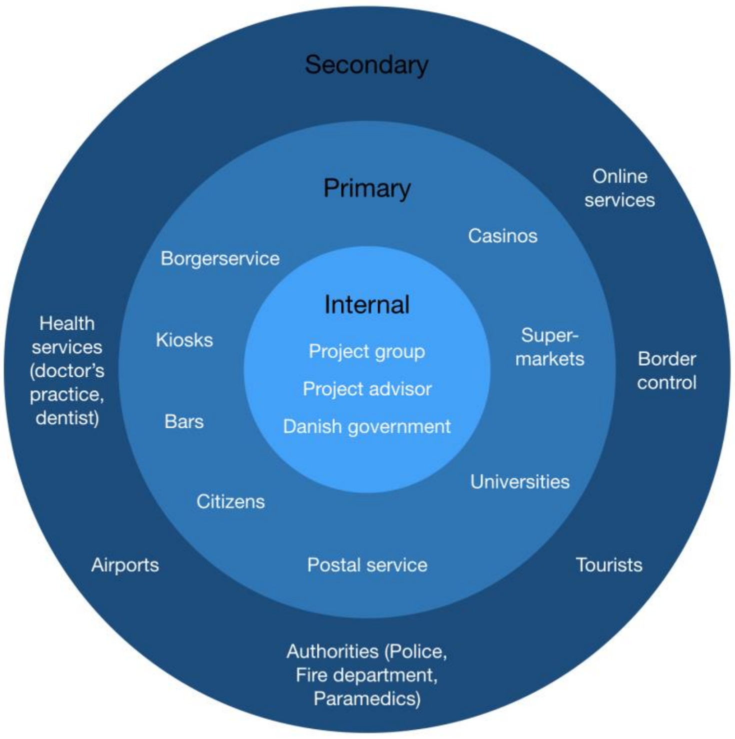


Figure 27: Stakeholders

Aim

The Project aims to develop a mobile application to replace physical ID cards. The solution will incorporate a secure access level system and server-side verification, which will determine which external users will be able to access certain information (i.e. name, age, picture, etc.).
Businesses will be able to utilize this application and setup accounts which they can modify in order to request different types of ID such as age, address, name, picture, etc.

Example

When going to the postal office to pick up a package, the store clerk requests ID such as a passport, driver's license or a medical card, in order to verify the customers identity. By doing so, the customer unnecessarily exposes personal information such as their personal identification number. With this application, using an account setup to fit the needs of the store, the store clerk will instead be able to request specifically the name of the customer, which the customer will then be able to accept or reject.

Objectives

1. Server-side verification
2. Modular design for 3rd party integration and expansion
3. Low bandwidth requirement for the app to work
4. Server-client architecture
5. Easy and user-friendly customization for business owners
    a. A store owner should easily be able to setup accounts for their employees to use, and easily customize which information the employees is able to request from customers.
6. Modifiability
    a. Analyze elements which might change in the future.

Ideas for solutions

For our solution to work properly, it will need to consist of several different software solutions. A mobile application will be used by the private users. They will use this app to log in and get their identification scanned. Store clerks will use a different application to scan the app the customers use. You could combine these functionalities in one single app; however, this would make the solution more complex than necessary, as customers would never use the clerk-functionality, and clerks would never use the customer-functionality. These apps will be developed in Android Studio with Java.
The app used by store clerks will need to be configured by company admins to fit the specific store it is used in. This is ideally done on a website where the admin can choose what information the clerks are able to scan off the customers' apps. Furthermore, DigID will also have an Internal Client for the internal stakeholders to use. Here, new companies can be added to the system. These websites will be created with the frontend framework Vue.
These applications will use a backend server which handles almost everything. When the mobile apps are used, they will contact the server through HTTP which handles the request. It is important that the verification happens server-side, as the client can easily be tampered with. The server will be developed in ASP.NET Core and will run in a Docker container.
If this was a real-world project, the system would need to communicate with NemID to retrieve the social security numbers of its users. As this is not possible in this project, we need to construct a replica of the NemID system to help pretend how DigID would look like in the real world. The same applies for the "Borgerservice"-part of the system. In the real world, DigID would use Borgerservice to look up social security numbers to retrieve information about the concerned citizen. Borgerservice will also handle the creation, deletion and alteration of DigID users. This is done through a website for the Borgerservice employees which is also created with Vue.
Ideally, we would like to extend our solution to enable use by third party developers. This could be useful when someone decides they want to use our identification system in their own way. For example, you could use this identification system to unlock doors at a facility instead of making employees carry identification cards.

Initial requirements analysis

Use case diagram

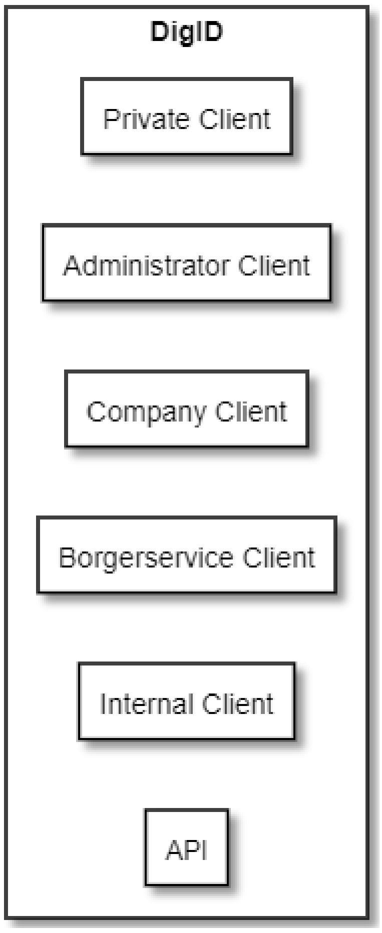


Figure 28: Use Cases overview

Individual use case diagrams can be found in *Appendix 1*

MoSCoW usecases

IC: Internal Client
CC: Company Client
BC: Borgerservice Client
PC: Private Client
AI: API Integrator
AC: Administrator Client

ID	Requirement	MoSCoW
IC-1	Sign In	M
IC-2	Setup new company	M
IC-3	Remove company	M
CC-1	Sign in	M
CC-2	Unlock	S
CC-3	Request ID	M
BC-1	Register photograph for private user	M
BC-2	Delete private user	M
PC-1	Sign in	M
PC-2	Unlock	S
PC-3	Start transaction	M
AI-1	Make API call	C
AC-1	Create ID controller account	C
AC-2	Alter ID controller account	C
AC-3	Remove ID controller account	C
AC-4	Adjust App configuration	C
AC-5	Get statistics	C
AC-6	Login	M

Domain model

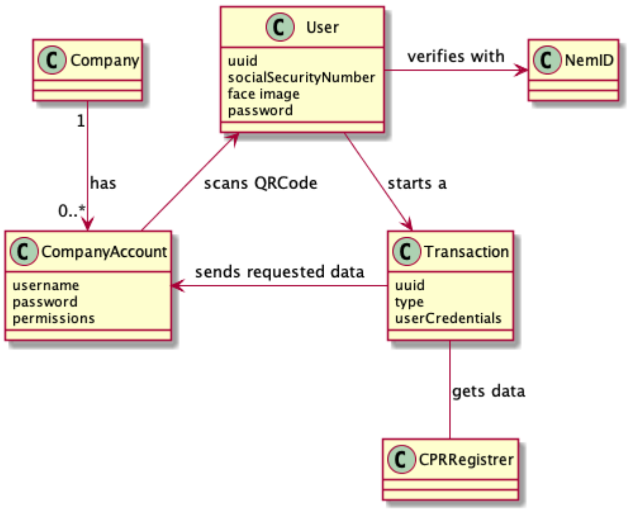


Figure 29: Domain Class Diagram

When an interaction between a store clerk (CompanyApp) and a customer (UserApp) happens, the store clerk will scan a QR Code on the customer's app, which will start at transaction which communicates with CPRRegistrer and sends data back to the store clerk which will inform the store clerk if the citizen is old enough to buy a pack of cigarettes for example.

Non-functional requirements

ID	Description	MoSCoW
F00	digID must not take longer than 1 second to open and be ready for a transaction to happen	S
F01	digID must not take longer than 3 seconds to handle a transaction between a company and a user	S
F02	The system must encrypt all stored data	M
F03	Users can try to log in five times before being locked out for 15 minutes.	S
F04	Users' needs to have their picture taken at an authorized place	M
F05	The servers must be able to handle at least 5% of all registered users at the same time	S
F06	Citizens should not be allowed to change any of their information with the application	M
F07	The application must work on Android	M
F08	Users must accept terms of service before using the application	M
F09	The system must be distributed	M
F10	The Mobile Application must work with slow internet connection	S
F11	The Mobile Application must be energy efficient	C
F12	The server must be developed with a modular design for 3rd party integration and expansion	C

Existing solutions

Smart-ID is an identification application made primarily by Danske Spil, with support from other companies. The application allows for a user to login with their NemID and create the digital Smart-ID based on the government's information about a citizen. It uses a picture taken by the user on the phone as the image showcased on the digital ID. The Smart-ID showcases an image, name, birthdate and age. The age field is colored based on whether a person is above a certain age. It only uses the client app, and has no other validation, which differs from how we plan to make our own system, and instead uses an animated background and an animation when the image is pressed to mitigate manipulated images being used to counterfeit the Smart-ID. It has met some criticism because of the user-inputted image.[11] The system also doesn't have any publicly know API's, nor any removal of information based on what purpose the identification has, contrary to our plans for a system.

Related research

The following section presents related information that is relevant for the project. It focuses on similar existing or upcoming projects. In this project we can learn from similar projects, decisions they have taken, and feedback they have gotten from the market.

Digital ID in Estonia[12]

The Estonian government claims they have the most developed national ID-card system in the world. Their physical card has a much broader list of use cases than the Danish health card ("sygesikringskort"). It has a supplementary mobile app (Mobile-ID) with unique security solution embedded inside the user's sim card. They have recently released another mobile app (Smart-ID) which

[11] https://politiken.dk/forbrugogliv/forbrug/art5931214/Test-af-digitalt-id-kort-Man-kan-nemt-snyde-med-det-og-det-har-startproblemer

[12] https://e-estonia.com/solutions/e-identity/id-card/

serves the same purpose but doesn't enforce a special sim card. The use cases of the mobile application however seem very similar to the Danish NemID mobile app.
Post on **Finansministeriet[13]: Digital driving license in progress**
The Danish government is planning to make all issued ID's digital, and the first step is making a mobile app for the Danish driving license. They point out not only the conveniences of not carrying physical cards, but also the reduced cost in administration that it would benefit.
The mobile app is expected to be released before the end of 2020.
Article published in **Politiken**[14]: Testing digital id card: It's easy to cheat and they have startup issues
In this article the author puts focus on the existing solution Smart ID and shares the first impressions. The app is generally seen as a good initiative, and the end-users agree that it is very easy and convenient and will save them the struggle of carrying their physical ID card. However, they point out a few issues with the app. They have had problems with not being able to login for a few days because of a security feature when switching ID. They also point out, that it's very easy to cheat and use someone else's ID.
The creators behind the app point out that their product is a beginning of a change in habits, and that there is need for more than functioning mobile in order to be successful with their idea.

Methods

- Scrum
  - Work will be structured with scrum; though it will have a few scrum buts such as:
    - We will be using weekly scrum instead of daily scrum as we will be coordinating work on a weekly basis instead of daily
    - We won't be having a specific project owner either as our product owner has a hypothetical owner
  - We will be ordering our sprints and our epics with the tools available in Azure DevOps
- UML
  - We will be using UML as a tool to structure the work, ensure that everyone has the same image of the plans and later to document the project
- UP
  - We will be working with Unified Process (UP) as a tool to help structure our workflow and to help determine the artifacts that we will be producing

Risks

1. The integrity of the system hinges on the security of the system, as such if the security is subpar, the system will fail. Hence any decision we make, must take security into account. This means that we must find secure and thoroughly tested cryptographic protocols and algorithms. In order to mitigate this risk, we must gain knowledge about security to a point where the team can make enlightened and smart decisions about security.

[13] https://www.fm.dk/nyheder/pressemeddelelser/2018/09/digitalt-koerekort
[14] https://politiken.dk/forbrugogliv/forbrug/art5931214/Test-af-digitalt-id-kort-Man-kan-nemt-snyde-med-det-og-det-har-startproblemer

2. If the systems are not updatable in an easy and fast fashion, the system would be deemed insecure with the first security hole that would be found, as it cannot be mitigated quickly. This would lead to distrust and migration to other solutions. We can mitigate this by designing the system with quick updates in mind, that forces the user to patch vulnerabilities in an unobtrusive way.
3. The application must work on phones, or else the system is pointless. Seeing as we have no experience with mobile-application development, this may e nd up being more work than anticipated for the team. We will mitigate this by using tool for the development of the apps that are closely related to already familiar tools.

Project Organization

Our goal is to spread work equally across the team members, this includes both implementation of code and writing report. This is done by delegating responsibility for specific topics and sections to different team members to make sure that all topics are covered widely. This does not necessarily mean that the respective member is singlehandedly going to complete this topic or section. However, it is their responsibility to make sure that somebody is making it.
We are using Azure DevOps and Microsoft Teams, to help manage our project and as a communication tool.
As a part of Unified Process and Scrum we are going to work in Sprints. Frederik will be Scrum Master and is therefore responsible for task creation and updating in Azure DevOps.

Milestones

- Project foundation
- Project Seminar video
- Finished Software prototype
- Finished report

Project plan

The project foundation is the first milestone and will be the first step. This includes our domain model, use case diagrams, research on existing solutions, risk assessments and an overarching plan for the project.
The project seminar video will showcase our progress with the project and will show a functioning database, functioning communication, and a basic user interface. The video will also show a deployment model for our solution as well as a domain model.
The finished software prototype will contain a database for storing relevant data, a server to deal with verification, data handling and access control, an app for citizens and an app for the user that needs to check ids. If we have the time required to do so, we will also be making an API for integrating the solutions into other third-party solutions.
The finished report will be a deliberation on our process. It will outline the thoughts we have had throughout the process of designing the software solution.

Deliverables

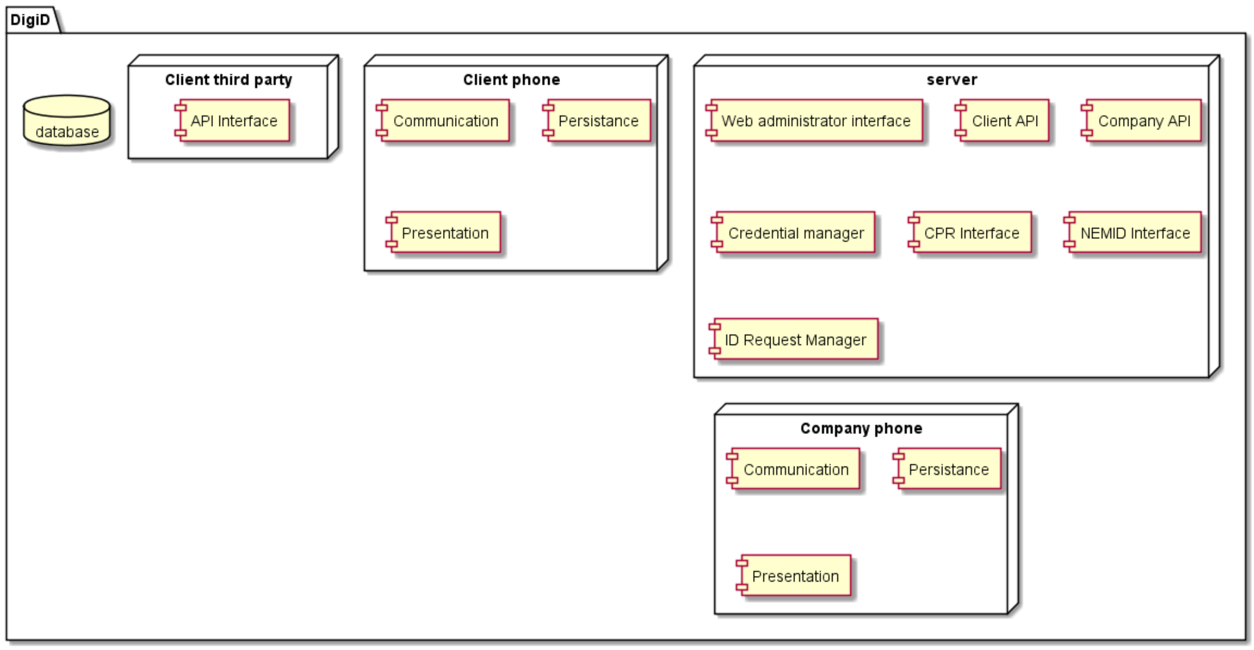


Figure 30: Deplayment model

Tentative outline for project report

The report will follow the guidelines in chapter 8 ”Presenting your project in written form” In (Dawson, 2015) [15]. In addition, we are planning to write about the following topics:

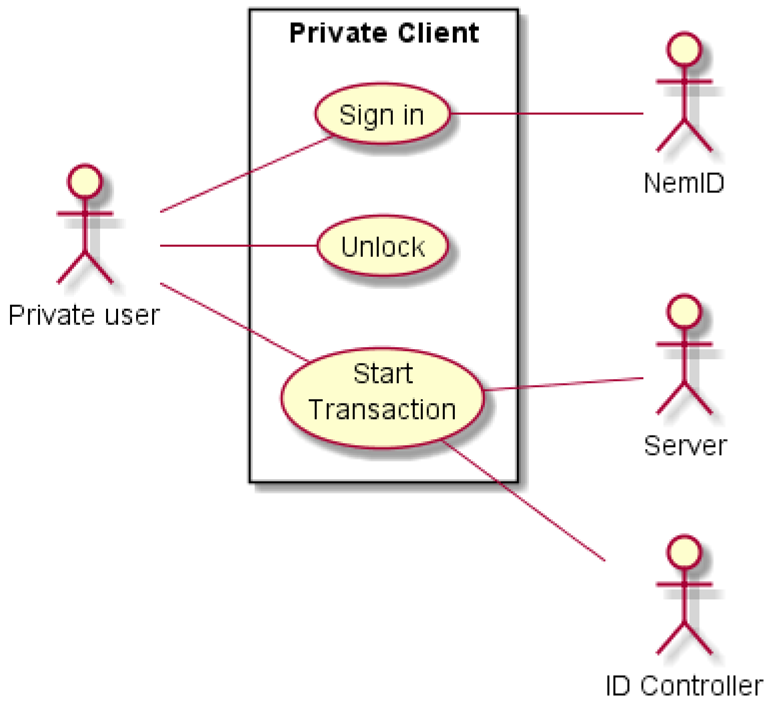
- Security
    - Explaining how we cover CIA triad
    - Threat Modeling
- Server-client
- Existing solution mistakes
- Spring-boot

[15] Projects in Computing and Information Systems 3rd edition, Christian Dawson, 2015

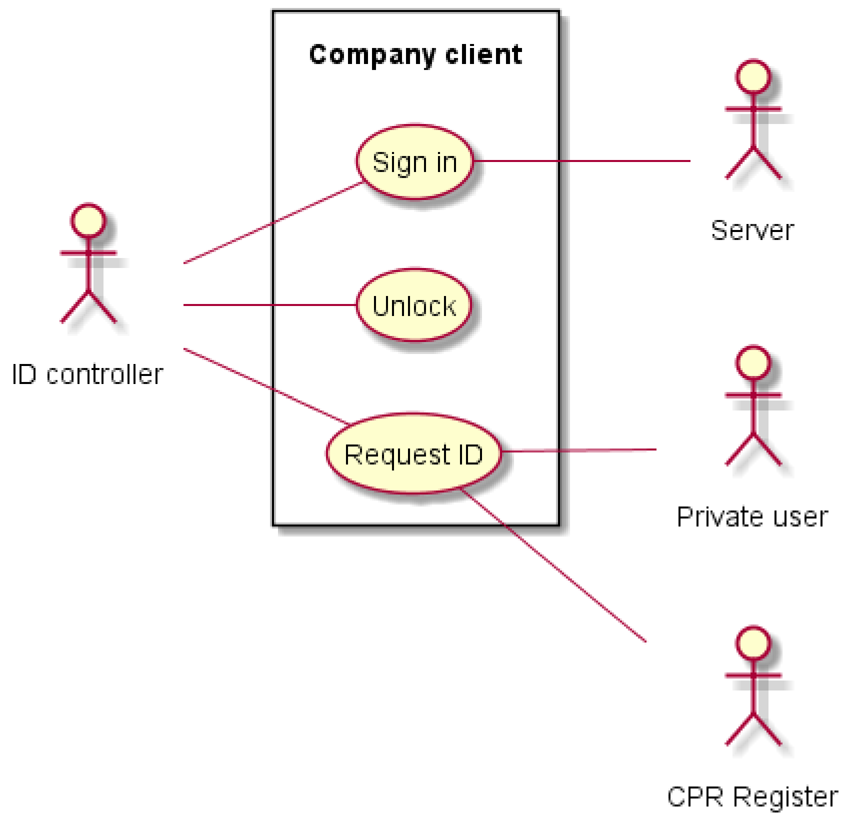
Appendix

Appendix 1: Use case diagrams

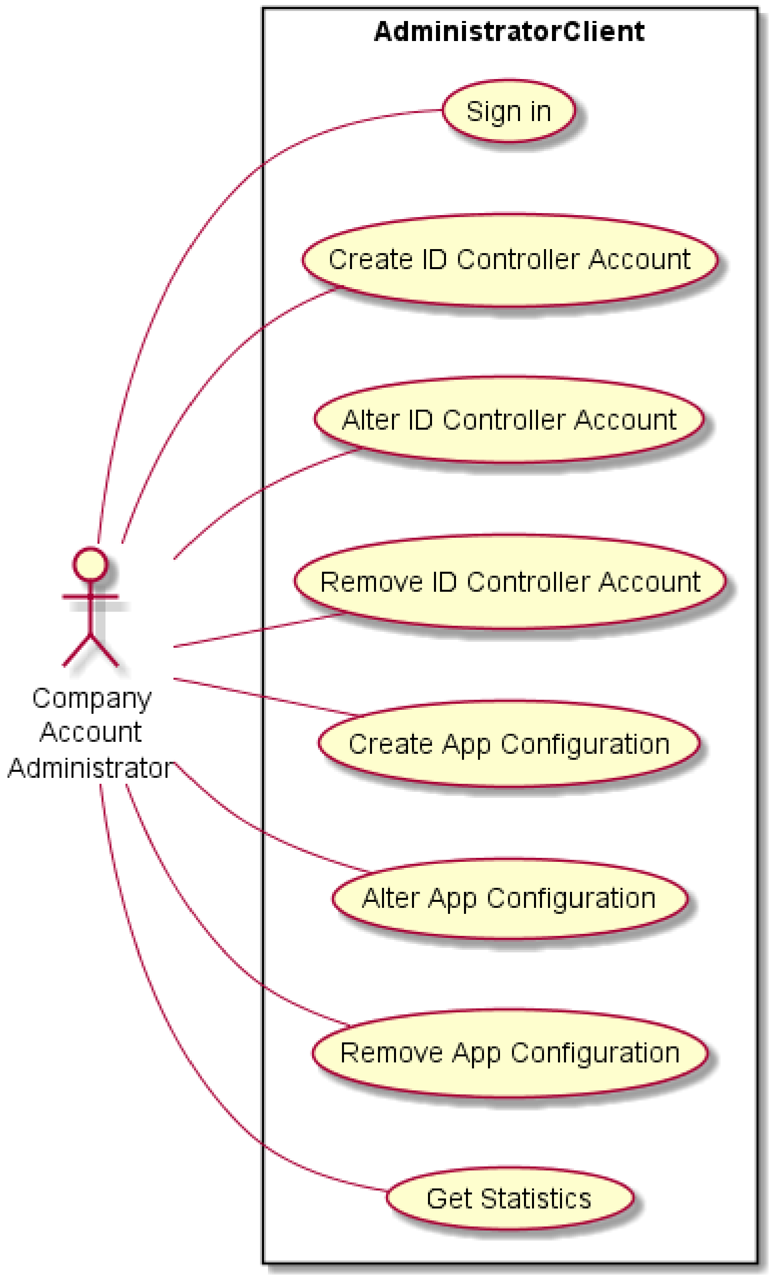
User Client use cases



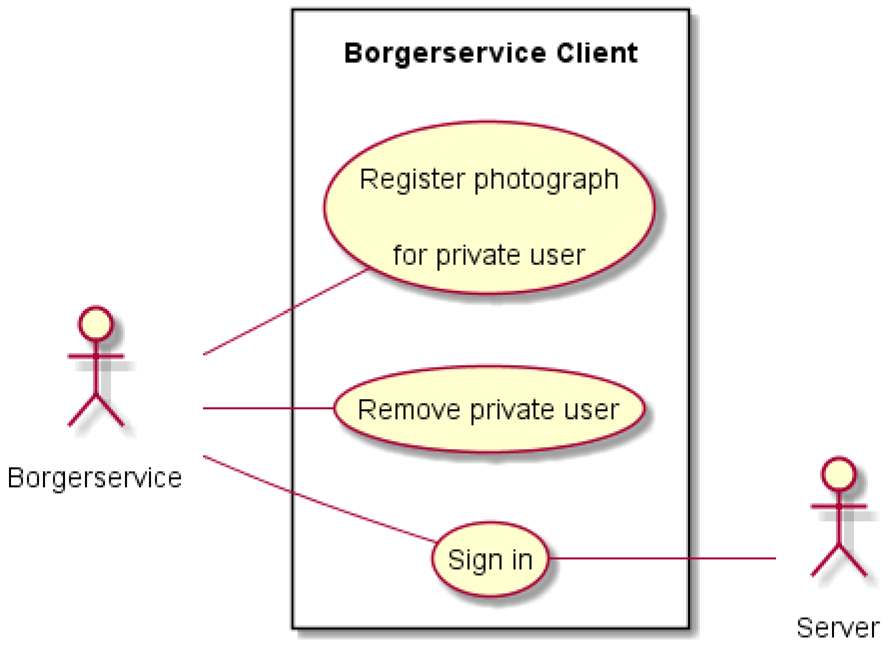
Company Client use cases



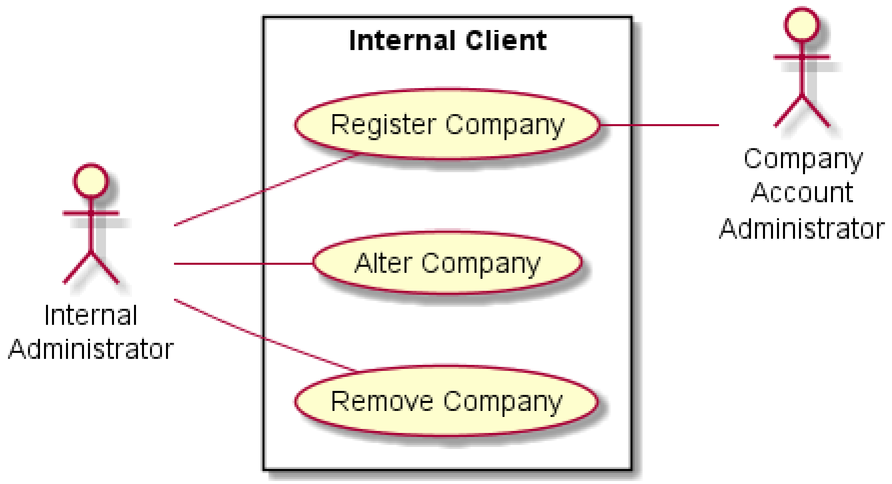
Administrator Client use cases


AdministratorClient
Sign in
Create ID Controller Account
Alter ID Controller Account
Remove ID Controller Account
Create App Configuration
Alter App Configuration
Remove App Configuration
Get Statistics
Company Account Administrator

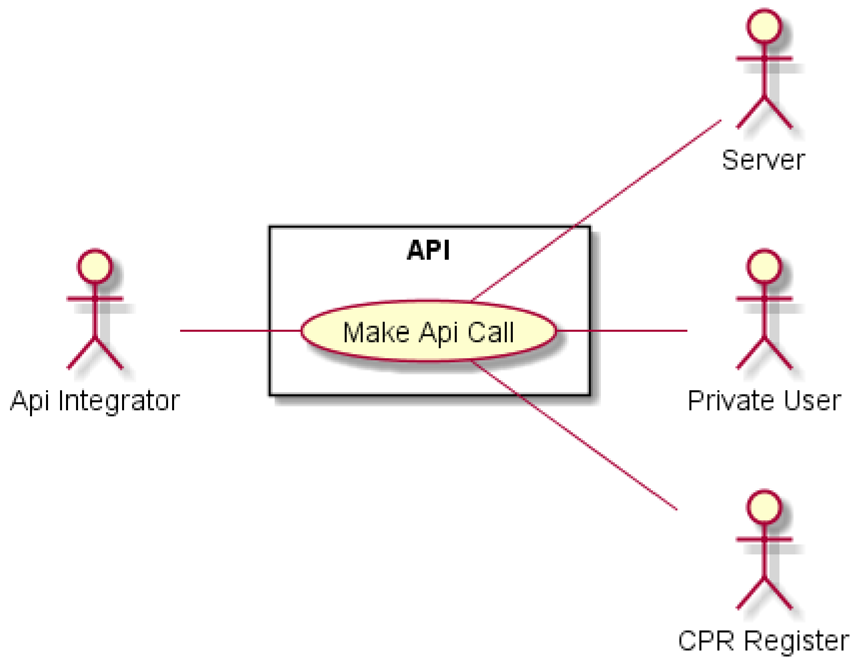
Borgerservice Client use cases



Internal Client use cases



API use cases


API
Make Api Call
Api Integrator
Server
Private User
CPR Register