



UiA University
of Agder

VIO

Group 0022

written by

Nancy Rønqist Erichsen

Adam Hazel

Gunn Marita Jomås-Britten

Nora Lior

in

IKT201-G 24H

Internet Services

With guidance from Christian Auby

Faculty of Engineering and Science

University of Agder

Grimstad, December 2024

Abstract

The VIO project aims to address the challenges faced by community organizations in managing memberships, communications, and financial operations. These organizations, ranging from small local groups to central entities, often rely on fragmented systems or manual processes, leading to inefficiencies and limited scalability. The need for a more cohesive, flexible platform is evident, particularly for smaller organizations that need to handle a diverse set of tasks, from event management to role-based communication.

To solve this problem, VIOUSE is being developed as a role-based platform that tailors functionalities based on user roles, such as central administrators, local group leaders, contributors, and members. The platform focuses on simplifying administrative processes while offering secure financial management, integrated communication tools, and scalable membership handling, including family accounts. By streamlining these processes, the platform enhances the overall efficiency of community organizations.

The solution provided by VIOUSE centers around its key features, such as membership registration, event scheduling, payment handling, and secure communication channels. The anticipated result is a versatile, user-friendly platform that allows organizations to manage their operations effectively while reducing administrative burden. This approach also supports scalability, making it suitable for both small and growing organizations.

In conclusion, VIOUSE offers a comprehensive solution that meets the needs of community organizations by improving their operational efficiency and fostering better engagement among members. Its role-based structure and focus on secure data handling make it a valuable tool for organizations looking to streamline their processes and better serve their communities.

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	Vi erklærer videre at denne besvarelsen: <ul style="list-style-type: none">• Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands.• Ikke refererer til andres arbeid uten at det er oppgitt.• Ikke refererer til eget tidligere arbeid uten at det er oppgitt.• Har alle referansene oppgitt i litteraturlisten.• Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).

Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Preface

This project, titled *VIO*, was developed as part of the third semester in the computer engineering course IKT201: Internet Services at the University of Agder. The goal was to design a scalable web application for managing memberships, payments, and local group administration, using the Scrum framework for collaboration.

The project was carried out by Nancy Rønqist Erichsen, Adam Hazel, Gunn Marita Jomås-Britten, and Nora Lior, with work conducted both at UiA Campus Grimstad and remotely. Tools such as Git, Jira, and Figma were used to support development and design.

We extend our gratitude to our lecturer, Christian Auby, for his guidance during biweekly meetings.

Grimstad 13. December 2024

Nancy Rønqist Erichsen, Adam Hazel, Gunn Marita Jomås-Britten, and Nora Lior.

Contents

1	Introduction	1
2	Limitations	3
3	Theoretical background	4
3.1	Local Group Structure	4
3.2	Central Organization and Local Groups	5
4	Method	6
4.1	Scientific Method	6
4.2	Project Timeline	6
4.2.1	Idea Phase	7
4.2.2	Planning	7
4.2.3	Core Development and Advanced Features	7
4.2.4	Final Adjustments	8
4.2.5	Testing and De-Bugging	8
4.2.6	Reporting	8
4.3	Process	8
4.3.1	Scrum	9
4.4	Role Allocation and Responsibilities	9
4.4.1	Workload Distribution	9
4.4.2	Meetings and Sprint Planning	10
4.4.3	Version Control and Collaboration Tools	10
4.5	Research	10
4.5.1	Competitive Analysis	11
5	Research Results	12
6	Requirements	15
6.1	Registration	15
6.2	Central Organization	15
6.3	Private User	16
7	Solution	17
7.1	Entity Model	17
7.2	Architecture and Framework	17
7.3	Design Principles	18
7.4	Product	19
7.4.1	Registration	19
7.4.2	Central Organization	20

7.4.3 Private User	20
8 Technical background	25
8.1 ASP.NET and MVC	25
8.2 Entity and Identity framework	25
8.3 API	25
8.4 AJAX	26
8.5 SMTP	26
9 Implementation	27
9.1 Architecture	27
9.2 Database and Models	27
9.3 Registration	28
9.4 Central Organization	31
9.5 Private User	32
10 Testing and Validation	54
11 Discussion	55
11.1 Process	55
11.2 Product	55
11.3 Implementation	56
11.4 Challenges	57
11.5 Methodological Reflection	57
11.6 For the future	58
12 Conclusion	60
Appendix A Product Vision	63
Appendix B Group Contract	63
Appendix C Product Info	63
Appendix D Time Sheets	63
Appendix E Meeting Minutes	63
Appendix F Demo Video	63
Appendix G Git Shortlog	64
Appendix H Sprint Reports	64

Appendix I	Individual Reports	64
Appendix J	Requirements	64
Appendix K	Figma Views	64
Appendix L	Entity Model	64
Appendix M	Test Report	64
Appendix N	Flow Diagram for Get Person	65

List of Figures

1	Organization structure for local groups	4
2	Organization structure for a central organization	5
3	Gantt chart of planned timeline	7
4	Key Results of Competitive Analysis	12
5	The MVC design pattern illustrated	18
6	View of solution: Initial User Registration	19
7	View of solution: Private User Registration	20
8	View of solution: Organization User Registration	20
9	View of solution: Organization User's Local Groups	20
10	View of solution: Adding Local Group	20
11	View of solution: Private User Details	21
12	View of solution: Persons connected to Private User	22
13	Adding Person	22
14	View of solution: Finding Available Local Groups	22
15	View of solution: Becoming a Member of a Local Group	22
16	View of solution: Private User Local Groups	23
17	View of solution: Local Group Info	23
18	View of solution: Membership Record	24
19	View of solution: Private User Payments	24
20	Sequence diagram of the registration process	29
21	Confirmation e-mail sent with Gmail	30
22	Interaction diagram for adding an admin to a local group	31
23	Relationship between ApplicationUser and PrivateUser	33
24	Interaction diagram for searching local group	34
25	Relationship between a PrivateUser and Person	35
26	Sequence diagram that represents the calls between services to add a person. Database is not included (accessed through services)	36
27	UserPersonConnection table showing connections between private users and persons	36
28	Flow diagram for implementation of transfer person	38
29	Modal with information presented to user before deleting user person connection	40
30	View of solution: Finding Available Local Groups	41
31	Overview of local groups in our implementation	41
32	Relationships between Person, LocalGroup, Membership and MembershipType models	41
33	View of the implemented "Mine lokallag"	45
34	Set local group to active	46
35	Entity model, Membership type	47

36	Editing membership type	49
37	Flow diagram of how to export membership list	50
38	Interaction diagram of how to export membership list	50
39	View of solution: AdminGroupMembers.cshtml	51
40	Outstanding Payment	53
41	Payment Processed	53
42	All test cases	54

List of Tables

1	Registration Requirements	15
2	Central Organization Requirements	15
3	Private User Requirements	16

Listinger

1	Dependency injection in CentralOrganisationController	27
2	AdminCreator class with validation attributes for admin assignment	28
3	JavaScript fetch request for partial registration with token validation	28
4	Configuration settings for email SMTP server in appsettings.json	30
5	LocalGroupAdmin entity defining relationships between local groups and users	32
6	Validation logic for ensuring a postcode contains only digits	32
7	JavaScript fetch request for sharing a person in the system	37
8	Validation to ensure the email belongs to a registered private user	37
9	Code snippet used in updating HTML after user successfully shares a person	37
10	ResultOfOperation class to standardize operation outcomes	39
11	Example of using the ResultOfOperation type in a method for deleting user-person connections	39
12	Membership entity defining relationships and properties	42
13	Code displaying the private function addMembership	43
14	Code displaying redirecting through AJAX	44
15	Code displaying the view models used for local group overview	44
16	Code snippet of view for "Mine lokallag"	45
17	Code snippet of using a service (_os) to generate the PersonOverview	45
18	Code snippet HTML code used so that a user can click and enter group page	46
19	Code displaying how buttons were implemented using Anchor Tag Helper	47

20	Code to display converting numbers into month names.	48
21	Code to display setting input as read only.	48
22	Code displaying how rows are sorted in JavaScript	51
23	Populating the PaymentViewModel from PaymentController	52
24	Code to display a pay button	53

1 Introduction

"Lokallag" is the Norwegian word used to define a local group consisting of members who share a common interest or purpose. Local groups are often connected to central organizations (also known as umbrella organizations). Due to the organic and fluid nature of local groups, membership management can often be inefficient or consume a large amounts of time. There is no predefined manner for handling of membership payments, membership lists or other administrative tasks.

In response to this challenge, this report presents a web-based solution for management of memberships in local groups located in Norway. The solution seeks to simplify and centralize key administrative tasks, whilst providing a simple-to-use interface for people who engage with one or multiple groups.

Though this project has been developed to be delivered as an exam for the subject "Internet services", led by Christian Auby at UiA Grimstad, the solution addresses real-world needs that could benefit many central organizations and local groups.

The following questions will be addressed:

- What are some key identifying factors of the domain for which this web-solution was created?
- What are the functional requirements that the web-solution needs to meet?
- What did the process (from conceptual design to implementation and testing) look like? And what were some of the key decisions that were made that influenced the building of the solution?
- What are the weakness of the implemented solution, and what steps could be taken to future development?

Product vision

VIO simplifies community organization management by providing tools for administration, communication, and financial handling tailored to roles such as admins, contributors, and members. Target customers include central organizations, local groups, and companies managing memberships and fees. The platform addresses needs like centralized and local group management, role-based communication, scalable membership handling (e.g., family accounts), and event/calendar management. Key attributes include secure financial management, GDPR-compliant data privacy, multi-user support, and a user-friendly interface. VIO stands out with its free, ad-supported, open-source model, multi-group support for admins and users, and a user-centric design enabling individuals to easily discover and connect with local groups.

A full product vision can be found in Appendix A, as well as directly here: *Product Vision*

2 Limitations

This project, while addressing the primary objectives of creating a membership management system, encountered several limitations during its development. These limitations, summarized below, impacted the scope and the outcomes of the project:

1. **Time Constraints:** The project was conducted within a fixed timeline of 14 weeks, which restricted the extent of functionality and polish that could be implemented.
2. **Resource Limitations:** Due to limited financial resources, certain tools and services that could have enhanced the project's quality or efficiency were not accessible:
 - Premium development tools or APIs were avoided due to cost constraints.
 - Paid database hosting or advanced performance monitoring solutions could not be utilized, affecting scalability testing.
3. **Access to Real-World Data:** The project was conducted in an academic setting, which limited access to real-world user data and interaction:
 - Testing scenarios were simulated, meaning the product was not validated under real-world conditions.
 - Feedback from potential end-users, such as central or local organizations, was limited, potentially impacting the usability of the solution.
4. **Technical Constraints:** Certain technical challenges were faced due to the team's experience and the technologies available:
 - Implementation of features like (e.g., advanced security measures or cross-platform compatibility) was limited by the team's current technical expertise.
 - The system's scalability and performance under high user loads could not be extensively tested due to hardware and hosting constraints.

3 Theoretical background

As stated in Chapter 1, local groups can be connected to central organizations. A central organization can be any type of organization that is registered in "Brønnøysundregisteret" (a public agency responsible for managing public register or business and organizations)[6]. Organizations that have local groups can include, but are not limited to charities, youth organizations and interest organizations. As the vision, aim and structure of each organization is unique, it is then difficult to compare the local groups of one organization to the local groups of another organization; their purposes, activities and meeting points may widely differ.

It can, on the other hand, be argued that, regardless of the uniqueness of each local group, there are at least two attributes that each local group shares:

1. A local group has registered members
2. A local group has someone who functions as an administrator of that local group

3.1 Local Group Structure

Figure 1 illustrates the basic structure of a local group. Each local group has registered members who participate in the group's activities and contribute to its goals. A key aspect of managing a local group is maintaining the membership base, ensuring active engagement and handling communication with members. Additionally, the group has at least one administrator or leader responsible for coordinating the group's operations.

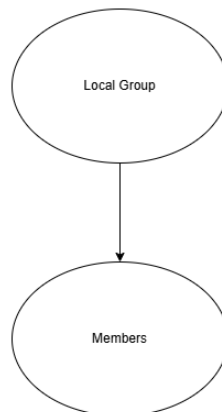


Figure 1: Organization structure for local groups

3.2 Central Organization and Local Groups

Figure 2 demonstrates how local groups connect to a central organization. The central organization provides oversight, resources, and strategic direction to its local groups, which operate semi-independently within their respective communities. Each local group, in turn, is responsible for its members and activities. This hierarchical relationship ensures alignment with the central organization's mission while allowing flexibility for local adaptations. As in organizations where the local group is not under a central organization umbrella, these local groups also have an assigned admin or leader, which manages the local group's member base as well as reports back to the central organization.

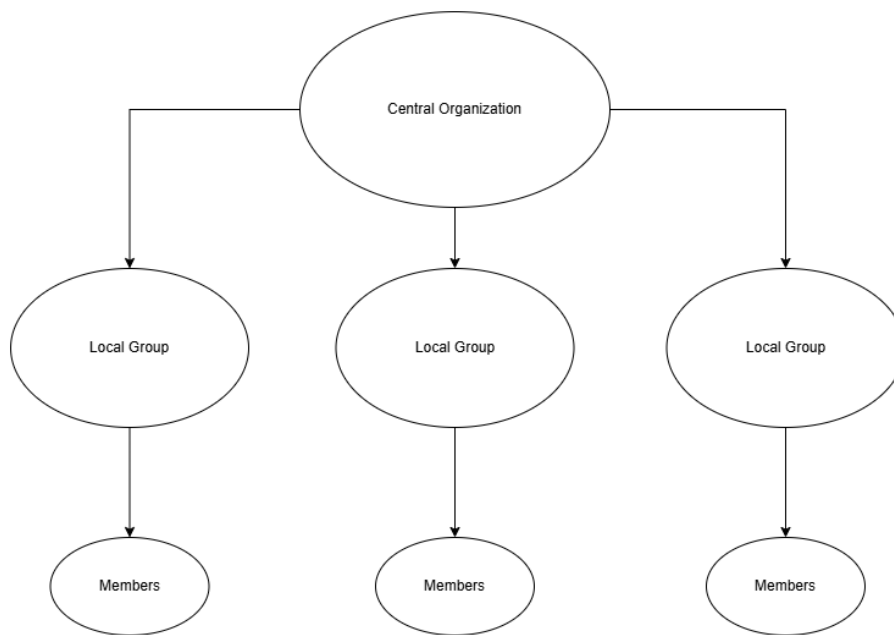


Figure 2: Organization structure for a central organization

4 Method

This section will describe the process to achieve the desired results of the project. A process provides rules and guidelines on who should do what, when, and why to reach a goal. The right process for a project depends on its challenges, such as the stability and complexity of the goal [22, p. 24]. An initial goal was set, but adjustments were made as limitations and time constraints became clear. This meant a plan-driven approach was used, while staying open to changes along the way with an agile process.

4.1 Scientific Method

The scientific method was incorporated into the project to ensure a structured and objective approach to evaluating existing solutions. The process began with an observation phase, during which competitors were identified and their general features and functionalities were surveyed to provide an overview of the competitive landscape.

Following this, the team proceeded to an information-gathering phase, focusing on collecting detailed data about the selected competitors. This included analyzing key functionalities, usability, and technical implementation. The collected information was systematically organized into a competitive matrix to facilitate structured comparison and analysis.

The analysis phase focused on identifying two key outcomes:

1. Essential functionality commonly found in other solutions that should be included in our project to meet user expectations.
2. Opportunities for differentiation, where unique features or improvements could set our solution apart.

This process ensured that our design decisions were grounded in a thorough understanding of the competitive landscape, balancing the need to align with industry standards while introducing innovative features to address unmet needs.

4.2 Project Timeline

The project was planned to span a 14-week period, beginning on September 2nd. The initial week was dedicated to brainstorming ideas and defining the project scope. This was followed by a five-week planning and requirements phase, where tasks such as user story creation and system design were prioritized. The next phase, focused on developing the core features for a minimum viable product (MVP), was allocated approximately five weeks. After completing the MVP, an additional two weeks were

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14
Idea Phase														
Planning														
Core Development														
Advanced Feature Development														
Final Adjustments														
Testing and De-Bugging														
Reporting														

Figure 3: Gantt chart of planned timeline

reserved for implementing advanced features. The final 3–5 days were dedicated to final adjustments and quality assurance, with product delivery scheduled for Friday, December 6th.

Testing and debugging were conducted iteratively throughout the coding process, ensuring early identification and resolution of issues. Additionally, reporting was an ongoing activity carried out throughout the project, starting from the second week.

4.2.1 Idea Phase

The initial phase of the project focused on selecting a concept to develop. Each group member was encouraged to propose project ideas, fostering a collaborative brainstorming process. From the suggestions, the group shortlisted two potential projects. A detailed discussion followed, during which the group analyzed the advantages and challenges of each idea. Based on these evaluations, the team reached a consensus on the most feasible and impactful project to pursue.

4.2.2 Planning

The second phase of the project focused on comprehensive planning, spanning a period of five weeks. With the project decided, the team began drafting user stories, use cases, design elements, system architecture, and an entity model. To minimize adjustments to the entity model and foundational system after coding commenced, the group allocated significant time to this phase. This deliberate approach allowed for a clearer understanding of how to initiate implementation and ensured a structured progression throughout the project.

4.2.3 Core Development and Advanced Features

The third phase was dedicated to core development, where the initial implementation of foundational components began. The focus was on developing essential features that would enable further progress in subsequent sections of the project.

For example, completing the user registration process was prioritized to support features requiring authenticated access. Tasks were distributed among team members, with weekly goals established to track progress. This phase culminated in the development of a minimum viable product (MVP).

Following the completion of the MVP, the team transitioned to the fourth phase, focusing on advanced features. These features, while not critical for the MVP, were intended to enhance the final product. Development in this phase prioritized functionality and usability improvements that aligned with project goals.

4.2.4 Final Adjustments

The fifth and final phase involved polishing the project and preparing it for delivery. Efforts during this phase included refining the user interface, resolving any outstanding issues, and ensuring that all components met the project requirements and quality standards.

4.2.5 Testing and De-Bugging

From the outset of the coding phase, the team adopted a systematic approach to testing and debugging. Each component was tested upon completion to ensure it performed according to requirements before moving on to subsequent tasks. This iterative testing process was documented to maintain a clear overview of functionality and identify any areas requiring improvement. The team established a group requirement that all tasks be thoroughly tested and debugged before they were considered complete.

4.2.6 Reporting

The reporting process commenced during the planning phase, where the team began drafting a preliminary version of the report. This draft served as a living document, updated continuously throughout the project to document progress, decisions, and challenges. This approach ensured that information was captured in real-time, streamlining the final phase of refining the report for delivery.

4.3 Process

At the outset of the project, the group developed a comprehensive plan to guide the development of the chosen product. While it was acknowledged that adjustments might be required along the way, the initial plan provided a structured overview of the project timeline and deliverables. The process began with the creation of user stories and requirements, followed by the development of use cases, a domain model, and an entity model to establish a strong foundation for implementation.

4.3.1 Scrum

Scrum was adopted as the primary framework for project management, incorporating key elements such as a product backlog, sprint backlog, and weekly sprints. Weekly meetings were held to review progress, create new sprint backlogs, and address challenges encountered during development.

Scrum is built around several components. The product backlog contains all potential work items for the project. During sprint planning, specific tasks are selected to form the sprint backlog, which guides the team's work for the duration of the sprint [22, p. 32]. At the conclusion of a sprint, the output—known as the product increment—reflects the completed functionality. Following each sprint, a sprint review is conducted to evaluate the product increment, and a sprint retrospective is held to identify opportunities for improvement in subsequent sprints [22, p. 33].

4.4 Role Allocation and Responsibilities

A group leader was appointed to facilitate meetings, document brief minutes, and manage updates in Jira. Tasks were divided weekly among team members, covering areas such as model development, requirements gathering, report writing, and software development.

All team members contributed as developers, implementing features with some agreed-upon coding guidelines to maintain consistency. Initially, task assignments were discussed during weekly group meetings, but later, sprints were introduced as a tool to organize work. One team member assumed the role of Git master, responsible for overseeing repository management and ensuring smooth branching and merging processes.

4.4.1 Workload Distribution

From the outset, it was agreed that each group member would dedicate a minimum of 108 hours to the project. To ensure accountability, the team logged all hours worked, with each entry documenting the date, duration, and description of the task.

Tasks were assigned based on the strengths and preferences of individual group members. For example, those with a particular interest in coding specific features or working on documentation were assigned tasks in those areas. While all team members contributed to each aspect of the project, leveraging individual strengths allowed for greater efficiency and progress.

4.4.2 Meetings and Sprint Planning

Weekly group meetings, typically lasting two hours, were held alongside bi-weekly meetings with the advisor. These meetings were conducted primarily on Microsoft Teams, as one group member resided outside Grimstad.

Each meeting followed a structured agenda. The team began by reviewing the previous week's progress and addressing any outstanding tasks. Discussions focused on resolving challenges and updating the backlog with new issues if required. The team also evaluated the project's progress against the timeline and adjusted plans as necessary to stay on track.

Sprint planning was a core component of these meetings. Suitable issues were selected for the upcoming sprint, and tasks were distributed among members. This process ensured clarity on individual responsibilities, minimized confusion during implementation, and reduced the need for ad-hoc coordination outside meetings.

4.4.3 Version Control and Collaboration Tools

Bitbucket by Atlassian was utilized as the remote Git repository for version control, integrated with Jira for task tracking. These tools, provided as a package by UiA, facilitated seamless collaboration and efficient branch management in alignment with user stories and active sprints.

One group member served as the Git master, overseeing branch merging and ensuring repository organization. The main branch was designated as the "last working version," while separate branches, labeled for each sprint (e.g., sprint1, sprint2), were used to safely integrate updates before merging into the main branch.

Additionally, Microsoft Teams was employed as the primary communication platform among group members. It provided a centralized space for discussions, virtual meetings, and file sharing, ensuring that all project-related resources were accessible and well-organized.

4.5 Research

Research was conducted as an integral part of the project's initial phases to ensure an informed and user-focused development process. The primary objectives were to evaluate existing solutions in the market, understand user expectations, and identify potential technical constraints. The research process included a review of similar tools, consultation of relevant academic and industry literature, and discussions within the team to contextualize findings.

These efforts helped inform the creation of user stories, technical requirements, and

design elements, ensuring that the project aligned with user needs and addressed gaps observed in existing solutions.

4.5.1 Competitive Analysis

A competitive analysis was conducted to evaluate existing solutions and gain insights into best practices and common challenges. Competitors were selected based on their relevance to the project's objectives and their prominence in the market. The analysis focused on criteria such as functionality, usability, scalability, and technical implementation.

The team documented observations using comparison tables and discussed the findings during planning meetings. This process provided valuable context for designing features and prioritizing functionality in the project.

5 Research Results

As part of the research phase, a competitive analysis was conducted to understand existing solutions in the membership management domain. This analysis focused on identifying the key features, strengths, and weaknesses of three prominent platforms: Consio, SmartOrg, and Cornerstone. The findings from this analysis are summarized below, with further details provided in the accompanying comparison chart.

	Cornerstone	SmartOrg	Consio	Spond
Target Customers	*Nonprofit organizations Religious communities and congregations *Youth and children's organizations	*Homeowner associations *Membership-based associations *House developers	*Medium to large organizations and associations *Trade unions *Sports teams and clubs *Non-profit organizations and volunteer groups	*Sports teams and clubs *Schools and educational institutions *Community groups and associations
Pricing	No data	From 39NOK per month	From 3400NOK per month	2NOK + 2,5% per transaction
Strengths	*Features for member engagement and attendance tracking *Strong customer relationship management	*Focused on homeowner associations and housing developers *Integrated accounting and event tools *No startup costs	*Comprehensive membership management *Tailored solutions for unions and large organizations *Advanced tools for tariff management and event planning	*Free to use *Excellent for sports teams and group communication *Simple payment integration via Stripe
Weakness	*Designed specifically for nonprofits and religious groups *No transparent pricing	*Less detailed information on customization or scalability	*Pricing is high	*Focuses more on coordination than robust administrative features *Payments include transaction fees
Competitive Advantage	Focused customer relationship management and engagement tools for nonprofits and religious organizations	Specialized systems for homeowner associations and housing developers with no upfront costs	Tailored membership solutions with advanced tools for unions and large organizations	Free, user-friendly platform ideal for sports teams and community groups

Figure 4: Key Results of Competitive Analysis

Below are detailed descriptions of the analyzed platforms, providing context for the features and findings outlined in the chart.

Cornerstone

Cornerstone specializes in providing solutions for central organizations, faith-based communities, and local groups organized under the Frifond model [10, 11]. The platform tailors its features to the specific needs of each focus area, with the Frifond modules offering tools highly relevant to this project's target audience.

Key features include member registration, end-of-year reporting, and SMS/email functionality for communication with group members.

SmartOrg

SmartOrg presents itself as a “simple and effective all-in-one solution for managing associations, groups, and housing cooperatives”, aiming to minimize administrative burdens [23]. The "SmartOrg Organisasjon" solution is particularly useful for central organizations, allowing them to manage both their own needs and those of connected local groups.

Pricing for SmartOrg appears to be determined based on organizational needs, and its offerings include a complete webpage solution, tools for managing organizational by-laws, and integrated accounting features.

Consio

Consio is marketed as a “highly functional and well-operating membership system”, designed to meet administrative and accounting needs for medium-to-large organizations and trade unions [7]. The platform positions itself as a partner, offering timely support, bug fixes, and system updates to its clients.

Consio offers a license-based pricing model, starting at 3400 NOK per month, with the final cost adjusted based on customer needs [9]. The system includes a wide range of modules, such as membership registration, membership benefits, and newsletter generation [8].

Spond

Spond is a free, user-friendly platform designed to simplify group coordination and communication. It is particularly popular among sports teams, schools, and community groups. The platform provides tools for managing activities, organizing events, and collecting payments through integrated systems [2].

What sets Spond apart is its free-to-use model, with revenue generated through transaction fees on payments processed via Stripe. This makes it an accessible option for smaller organizations or grassroots teams. However, its focus on group coordination rather than robust administrative tools may limit its appeal for larger organizations.

Analysis and Key Insights

The competitive analysis revealed several important trends in existing solutions:

- **Core Features:** All platforms prioritize membership registration, financial management, and communication tools, indicating these are essential functionalities for users.

- Pricing Models: While Consio and SmartOrg offer premium, license-based solutions, Spond stands out for its free-to-use model, making it accessible to smaller organizations.
- Differentiation Opportunities: Cornerstone focuses heavily on religious and youth groups, and Consio targets unions, leaving a gap for a flexible, cost-effective solution that serves smaller local groups or new users seeking a user-centric platform.

These findings informed the functionality and unique features prioritized in the development of this project. Specific design choices, such as the modular structure and cost-effective features, are detailed in Chapter 7 and Chapter 9.

6 Requirements

This section summarizes the core functional requirements necessary to deliver the Minimum Viable Product (MVP) for the VIO platform. These requirements were derived from user stories, assessed for priority and development difficulty using the Fibonacci scale:

- **1-3: Could-have** – Features that are desirable but not essential for the minimum viable product (MVP).
- **5: Should-have** – Features that add significant value but can be deferred if necessary.
- **8: Must-have** – Critical features required to meet core user needs and deliver the MVP.

The following tables present only the requirements prioritized for the MVP. A detailed description of all functional and non-functional requirements, including those de-prioritized during this project or considered for future development, is provided in Appendix J, as well as directly here: *Requirements*. Non-functional requirements and other contextual details can also be found there.

6.1 Registration

Table 1: Registration Requirements

ID	Requirement Description	Priority	Difficulty
UF1	Create an account to access system functionalities	8	3

6.2 Central Organization

Table 2: Central Organization Requirements

ID	Requirement Description	Priority	Difficulty
LGM1	Create local groups with assigned admins	8	3

6.3 Private User

Table 3: Private User Requirements

ID	Requirement Description	Priority	Difficulty
AF6	Manage multiple local groups with admin permissions	8	5
MM4	Block members	5	3
MM8	Export member records for end-of-year reporting	8	3
MM11	Manage multiple persons (e.g., family members)	8	8
MM12	Share a person with another user to manage memberships	8	8
MM13	Transfer persons to another user	8	5
MM17	See all persons connected to the account and their memberships	8	3
MM18	Cancel memberships connected to a person	8	5
MM19	Unblock members	5	3
UF2	View all available local groups	8	3
UF3	Register a person in a local group	8	3
UF4	Pay outstanding membership fees	8	8
UF5	Download receipts for payments	5	5
UF6	View details of previous or outstanding payments	5	3
UF7	Update personal details	8	2

7 Solution

This chapter presents the solution developed to address the specified requirements for the VIO platform. The solution, including design and functionality, aims to cover the requirements defined in chapter 6, ensuring alignment with the minimal viable product (MVP). All designs were created in Figma and serve as the foundation for the implementation.

7.1 Entity Model

The entity model illustrates the database structure used in the VIO platform, showing relationships between key entities such as users, organizations, memberships, payments, events, and messaging. The full diagram is available in Appendix L, as well as an interactive SVG file via the link below:

[Click here to view the full entity model.](#)

Key features of the entity model include:

- **Users:** Differentiates between private users and administrators. Each private user can manage multiple connected persons (e.g., family members) and their memberships.
- **Organizations and Local Groups:** Central organizations manage local groups, assign administrators, and oversee group-level operations.
- **Memberships and Payments:** Tracks membership details and links payments to individual or multiple memberships.
- **Events and Messaging:** Supports event participation tracking and a messaging system for communication between users and groups.

This model forms the foundation for the implementation of functionalities discussed in following chapters.

7.2 Architecture and Framework

The product will use the MVC (Model-View-Controller) design pattern. The application will be separated into three main areas [12], each with their own responsibility. The "Model" section handles business logic and data, the "Controller" (or better, controllers) handles requests from the user and communication with the Model, returning a view with which the user can interact with.

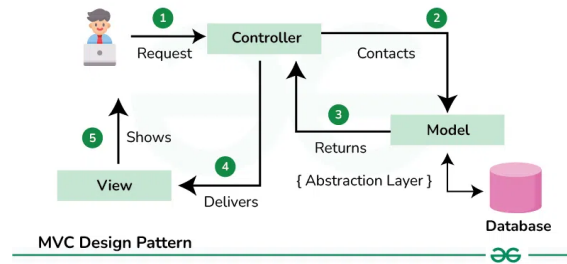


Figure 5: The MVC design pattern illustrated [12]

The product will utilize SOLID principles [1]. In short, the product should:

- be built using classes that are simple and focused, with only one reason to change
- allow for extension of classes without needing to change existing code
- neatly handle subclasses when necessary
- use simple, well defined interfaces that do not demand unnecessary implementation in the classes that realize them
- avoid highly coupled classes, and use interfaces when objects need functions found in other classes

The product will use ASP.NET Core framework. A key feature of this framework is that it has built-in support for dependency injection [13], allowing the programmer to avoid "hard-coded dependencies" between classes and thus provided an easy way to uphold the "D" of SOLID ("dependency-inversion-principle").

More details about the framework chosen for this project are described in the chapter *Technical Background*.

7.3 Design Principles

Some basic design principles will be considered during this project, including:

Visual Hierarchy: This principle helps users easily identify the most important elements on a page. This will be achieved by using colors and sizes to make key elements stand out[16].

Golden Ratio: Designs based on the golden ratio are believed to look better. This principle will be used to place menus and other elements in a way that looks nice and keeps the page organized and easy to use[16].

Hick’s Law: This principle says that the more choices a user has, the longer it takes to make a decision. To address this, the pages will have fewer options, making it simpler for users to find what they need[16].

White Space and Clean Design: White space is the empty space between elements like text, images, and columns. It’s not just blank space; it’s an important part of design. A clean design will use white space to make the page easier to read and understand, focusing on simplicity while still providing the necessary information[16].

7.4 Product

The following designs were created for the product to align with the requirements for a MVP.

A complete collection of Figma views, including designs for requirements that were deprioritized or not implemented in the MVP, is provided in Appendix K, as well as directly here: *[Click here to view the complete Figma Views.](#)*

7.4.1 Registration

In VIO, users can register without being a member of a local group beforehand. Users can create an account in the system and then find the local group they want to join through the website. This makes it easier for people to discover activities in their area that they wish to engage with.

The following views will cover UF1 - Create Account.

New users must first provide an email and password to create an IdentityUser, which requires email verification.

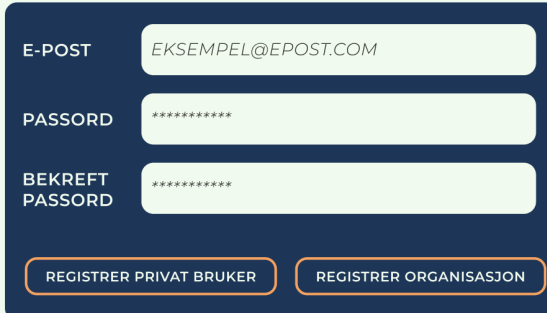
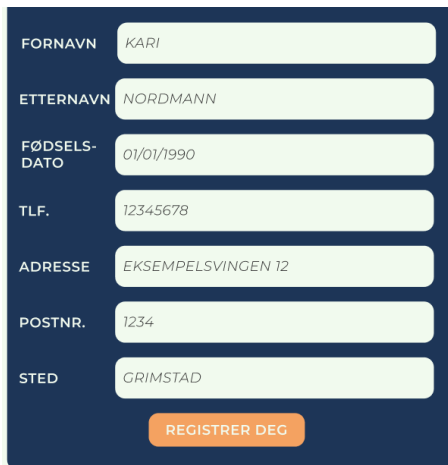
The image shows a registration form with a dark blue background. It contains three input fields: 'E-POST' with the placeholder 'EKSEMPEL@EPOST.COM', 'PASSORD' with a masked password '*****', and 'BEKREFT PASSORD' with a masked password '*****'. At the bottom, there are two buttons: 'REGISTRER PRIVAT BRUKER' and 'REGISTRER ORGANISASJON'.

Figure 6: View of solution: Initial User Registration

Registration will then branch to either private user registration or central organization registration to account for the information being stored in separate database

tables.



A registration form for private users with a dark blue background. It contains seven input fields with light green borders and placeholder text: FORNAVN (KARI), ETTERNAVN (NORDMANN), FØDSELS-DATO (01/01/1990), TLF. (12345678), ADRESSE (EKSEMPELSVINGEN 12), POSTNR. (1234), and STED (GRIMSTAD). At the bottom is an orange button labeled 'REGISTRER DEG'.

Figure 7: View of solution: Private User Registration



A registration form for organization users with a dark blue background. It contains two input fields with light green borders and placeholder text: ORG. NR. (123 456 789) and ORG. NAVN (MIN BEDRIFT). At the bottom is an orange button labeled 'REGISTRER ORGANISASJON'.

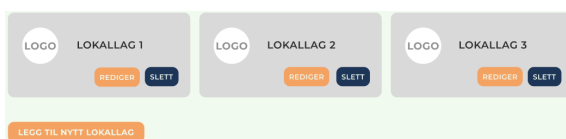
Figure 8: View of solution: Organization User Registration

7.4.2 Central Organization

"Mine lokallag"

A central organization user can add multiple local groups under their account. They will also have the ability to edit or delete any existing local group linked to them.

This view covers LGM1 - Create Local Groups.



A view showing a list of three local groups. Each group is represented by a grey card with a 'LOGO' icon, a name (LOKALLAG 1, 2, 3), and two buttons: 'REDIGER' (orange) and 'SLETT' (dark blue). At the bottom is an orange button labeled 'LEGG TIL NYTT LOKALLAG'.

Figure 9: View of solution: Organization User's Local Groups



A form titled 'LEGG TIL LOKALLAG' with a light green background. It contains six input fields with white borders and placeholder text: LOKALLAGSNAVN (EKSEMPELNAVN), ADRESSE (EKSEMPELSVINGEN 12), POSTNR. (1234), STED (GRIMSTAD), FYLKE (AGDER), and a 'LAGRE' (orange) button at the bottom.

Figure 10: View of solution: Adding Local Group

7.4.3 Private User

"Profil"

Private user will be able to view their profile, and edit their personal details, including their password.

This view covers UF7 - Update Personal Details

The screenshot shows a user profile page for Kari Nordmann. At the top left is a circular profile picture placeholder with the text 'REDIGER PROFILBILDE' below it. To the right of the profile picture is the name 'KARI NORDMANN'. Further right is an orange button labeled 'REDIGER PROFIL'. Below the name, there are several input fields for personal details: 'E-POST' (EKSEMPEL@EPOST.COM), 'PASSORD' (masked with asterisks), 'FØDSELSDATO' (01/01/1990), 'TLF.' (12345678), 'ADRESSE' (EKSEMPELSVINGEN 12), 'POSTNR.' (1234), and 'STED' (GRIMSTAD). To the right of the 'PASSORD' field is an orange button labeled 'ENDRE PASSORD'. At the bottom center is an orange button labeled 'LAGRE'.

Figure 11: View of solution: Private User Details

"Mine Brukere"

In VIO, a private user will have the option to add persons connected to them through their user profile. It will also be possible to share a person between two private users in the system or fully transfer a person to another private user. A private user can also de-register a connected person from a local group or completely remove them from the system.

This view covers MM11 - Manage Several Memberships, MM12 - Share private User Responsibilities, MM13 - Transfer private User Responsibilities, MM17 - See Memberships, and MM18 - Cancelling a Membership.

Figure 12: View of solution: Persons connected to Private User

Figure 13: Adding Person

"Lokallag"

Private users can easily search for and discover new local groups they want to join. This view covers UF2 - See Available Local Groups and UF3 - Become a Member in a Local Group

Figure 14: View of solution: Finding Available Local Groups

Figure 15: View of solution: Becoming a Member of a Local Group

"Mine lokallag"

Private users of the system will have an overview of the local groups they or persons connected to them are members of. If a private user is an admin of a local group they will also have an overview of those groups.

This view covers AF6 - Manage Multiple Local Groups.

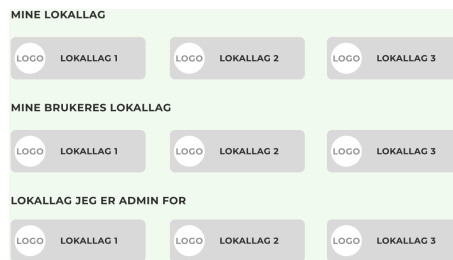


Figure 16: View of solution: Private User Local Groups

"Lokallaginfo"

A local group admin can edit the group's information, as well as toggling it to active or inactive. In addition to this, the admin can add what membership types the group in question offers.

This view covers AF6 - Manage Multiple Local Groups.

Figure 17: View of solution: Local Group Info

"Medlemsliste"

A local group admin can view an overview of the group's members. They can also block or unblock members.

This view covers MM4 - Block Member, MM8 - Export Record, MM19 - Unblock Member and AF6 - Manage Multiple Local Groups.

SØK ETTER MEDLEM: _____

☐ BETALT ☐ IKKE BETALT MEDLEMSKAPSTYPE: _____ ÅRSTALL: _____

[EKSPORTER MEDLEMSLISTE](#)

ID	NAVN	ALDER	ADRESSE	MEDLEMSKAPS -TYPE	MEDLEM SIDEN	MEDLEMS -STATUS	BETALINGS -STATUS

[BLOKKER MEDLEM](#) [OPPHEV BLOKKERING](#)

Figure 18: View of solution: Membership Record

"Betalinger"

Private users can easily access an overview of previously paid invoices. Additionally, the system provides the option to pay outstanding invoices.

This view covers UF4 - Pay Membership Fees, UF5 - Download Receipt and UF6 - See Previous and Outstanding Payments.

ID	NAVN	LOKALLAG	AKTIVITET	MEDLEMSKAPS -TYPE	PRIS	BETALINGS -STATUS	BETALT DATO	GYLDIGHETS -PERIODE	
1		GRIMSTAD TENNISKLUBB		FAMILIE	500NOK	BETALT	01.01.2024	01.01.2024-31.12.2024	(V)
1	KARI NORDMANN	GRIMSTAD TENNISKLUBB							
1	OLA NORDMANN	GRIMSTAD TENNISKLUBB							
1	LARS NORDMANN	GRIMSTAD TENNISKLUBB							
2	OLA NORDMANN	GRIMSTAD SJAKKLUBB		ENKELT	200NOK	BETALT	10.03.2024	01.01.2024-31.12.2024	
3	LARS NORDMANN	FK JERV		ENKELT	250NOK	IKKE BETALT		01.01.2025-31.12.2025	

[LAST NED FAKTURA](#) [BETAL](#) [ADMINISTER FAMILIEMEDLEMMER](#)

Figure 19: View of solution: Private User Payments

8 Technical background

8.1 ASP.NET and MVC

ASP.NET Core MVC is a framework to build web applications and APIs using the Model-View-Controller design pattern[24]. The MVC pattern has three main components: Models, Views, and Controllers. With this pattern, user requests are routed to a controller. Controllers handle user interactions, coordinate with model, and determine which view to render. The model represents the application's state and contains business logic or operations that need to be performed. View is responsible for presenting content to the user through the user interface. ASP.NET Core maps URLs to specific actions in the application. This allows you to create applications with clean, user-friendly, and searchable URLs[24].

8.2 Entity and Identity framework

ASP.NET Core Identity is an API that provides functionality for user management, authentication, and authorization. It includes features such as user interface components for login, user handling, password management, profile data, roles, email confirmations, and more[20].

Identity is typically configured with a SQL Server database to store usernames, passwords, and profile data. It also includes Account Creation and Login. Identity supports user registration, login, and logout functionalities. Identity integrates with ASP.NET Core, leveraging dependency injection to deliver its services efficiently[20].

Entity Framework allows developers to interact with data as domain-specific objects and properties, without needing to worry about the underlying database tables and columns. This enables developers to work at a higher level of abstraction, reducing the amount of code required to build and maintain data-driven applications[18].

The framework translates operations into database-specific commands, so that it's not needed to hardcode dependencies on particular data sources. Object-oriented programming often faces challenges when interacting with relational databases. This problem is often referred to as "impedance mismatch." Entity Framework resolves this by mapping relational tables, columns, and foreign key relationships in logical models to entities and relationships in conceptual models. This provides flexibility in designing objects while optimizing the logical model[18].

8.3 API

An API, Application Programming Interface is a set of definitions and protocols that enable communication between two applications. APIs facilitate the exchange

of data between systems, acting as a bridge between the user-facing system and back-end services[14].

There are three main types of APIs: Open APIs are public APIs with minimal restrictions, making them accessible for general use. Partner APIs require specific access permissions and are typically used for collaboration between businesses or partners. Internal APIs are hidden from external users and used exclusively within internal systems[14]. To interact with an API, standard protocols must be followed when making an API call. This ensures that communication between the systems is structured and predictable. There are three primary types of API protocols: REST, SOAP and RPC. REST is the most widely used API type. These are web service APIs that utilize HTTP for communication. They support operations such as Create, Read, Update, and Delete[14].

8.4 AJAX

AJAX, short for Asynchronous JavaScript and XML, is a web development technique that enables the creation of dynamic and interactive web applications. It allows a web page to retrieve data from a server asynchronously without requiring a full page reload, improving the user experience by making interactions faster and more seamless[4].

8.5 SMTP

SMTP (Simple Mail Transfer Protocol) is a protocol for sending and transferring email messages[15]. It operates by establishing a direct TCP connection between the sender's and receiver's mail servers, ensuring reliable delivery. SMTP uses a push mechanism to send emails and relies on commands like MAIL FROM and RCPT TO for communication. SMTP is the backbone of email systems, enabling users to send messages efficiently over the internet[15].

9 Implementation

9.1 Architecture

As stated in chapter 7, the product was to use the MVC design pattern, whilst paying special attention to SOLID principles. One way of fulfilling these principles was to use "services" as explained by Barbettini [5] in his book "Little ASP.NET Core Book". Rather than coding model functionality directly in a controller (for example, accessing the database directly from a controller), we used a service, accessible through its interface to carry out the needed functionality. An example is given in the code snippet below:

```
1 public class CentralOrganisationController : Controller
2 {
3     private readonly ApplicationDbContext _db;
4     private readonly ICentralOrganisationService _cos;
5
6     public CentralOrganisationController
7     (ApplicationDbContext db, ICentralOrganisationService cos)
8     {
9         _db = db;
10        _cos = cos;
11    }
12
13    // In an action:
14    var viewModel=_cos.GetCentralOrganisationByUser(userId);
```

Listing 1: Dependency injection in CentralOrganisationController

On a larger scale, we also implemented the idea of separation of concerns in the file structure by using Areas. As stated by ASP.NET Core documentation, "Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models"[25].

9.2 Database and Models

Models are descriptions of the entities stored in the database. As shown in the entity model in chapter 7, the product domain consists of entities that are connected to each other in some way. To ensure that these connections are upheld in our product, we used "fluent API" [3] to explicitly map some of the relations between our models, also defining what should happen to connected objects when one of those objects are deleted.

ASP.NET Core also provides "mapping attributes" also known as "data annotations"[3]. These were used to validate the correctness of user input when filling in

forms. Although a variety of annotations are provided, we also created custom annotations, thus moving business logic from being directly expressed in the controller. An example of a custom annotation being used in a model is provided below:

```
1 public class AdminCreator
2 {
3     [Required]
4     [PrivateUserExistsValidation]
5     [AdminAlreadyAssignedValidation]
6     public string AdminEmail { get; set; } = string.Empty;
7
8     public Guid LocalGroupId { get; set; }
9 }
```

Listing 2: AdminCreator class with validation attributes for admin assignment

9.3 Registration

In order to fulfill UF1, where a user can register, the registration page had to be split up into two separate views that would display based on the choices the user made in the registration process. It was essential to ensure that the first part of the registration process involved creating an *ApplicationUser* using the provided email address and password. This base user object would then serve as the foundation for either a private user or an organization user, depending on the selected option.

To achieve this, the system implemented a split-view approach for the registration page. When the user either clicks on "Register as private user" or "Register as central organization", an AJAX call is made to submit the initial data (email and password) and create the *ApplicationUser*. This ensures the page does not reload and allow the appropriate fields for either private user or central organization to dynamically appear for the user to fill out.

```
1 fetch('/Identity/Account/RegisterPartial', {
2     method: POST,
3     headers: {
4         Content-Type: application/x-www-form-urlencoded,
5         RequestVerificationToken: document.querySelector
6             (input[name="__RequestVerificationToken"]).value
7     },
8     body: formData.toString()
9 })
```

Listing 3: JavaScript fetch request for partial registration with token validation

In addition, another AJAX call was utilized to allow the user to confirm their email via a confirmation link without leaving the page. This ensures a smoother regis-

tration flow, enabling the user to complete the rest of the form seamlessly after verification.

The sequence diagram in figure 20 illustrates the flow of the registration process. Initially, the User interacts with the *RegisterPartial* controller, which handles the creation of an *AspNetUser* using *CreateAsync(string, string)*. Once the base user object is created, the process diverges depending on whether the user chooses to register as a private user or a central organization user.

If the user fills out the forms in the private user registration, and presses the "complete registration" button, the system invokes the *RegisterPrivateUser* controller, which validates the additional fields for private user and inserts the relevant data into the database. Similarly, filling out the forms in the central organization user registration and pressing the "complete registration" button, invokes the *RegisterOrganizationUser* controller, which performs the same steps.

By using this architecture, the registration process ensures modularity, smooth transitions between steps, and improved user experience by not reloading the page or redirecting to a different page view.

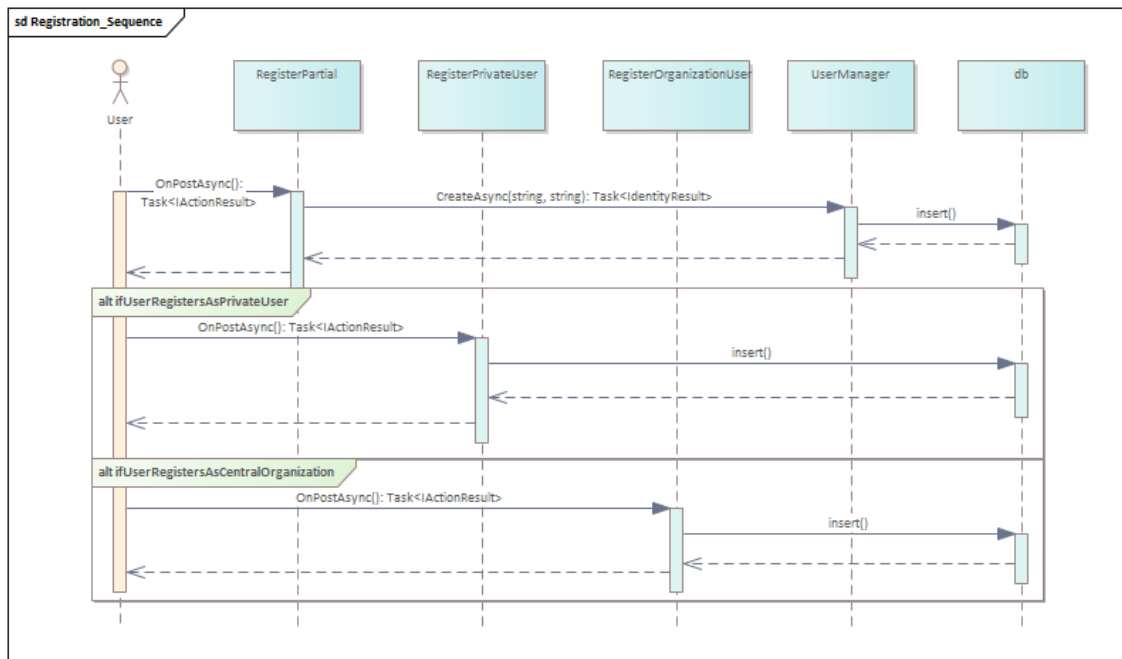


Figure 20: Sequence diagram of the registration process

E-mail confirmation

To enable email functionality, the project utilizes Simple Mail Transfer Protocol (SMTP) for sending emails, such as registration confirmations and password resets.

ASP.NET Core provides built-in support for email functionality, which we leveraged with Gmail's SMTP service to implement email confirmation and password reset. The configuration was stored in `appsettings.json` to manage the SMTP server details and credentials. This approach allowed us to use Google's SMTP service (`smtp.gmail.com`) with port 587 and ensure secure authentication for sending emails.

Below is an excerpt from the `appsettings.json` file, showcasing the configuration setup:

```
1  "EmailSettings": {  
2    "SmtpServer": "smtp.gmail.com",  
3    "SmtpPort": "587",  
4    "SenderEmail": "vionoreply@gmail.com",  
5    "SenderPassword": "qtowikloopomrezr"  
6  }
```

Listing 4: Configuration settings for email SMTP server in `appsettings.json`

This configuration enabled seamless integration of email services into workflows like user registration and password recovery, enhancing the user experience with automated email delivery.

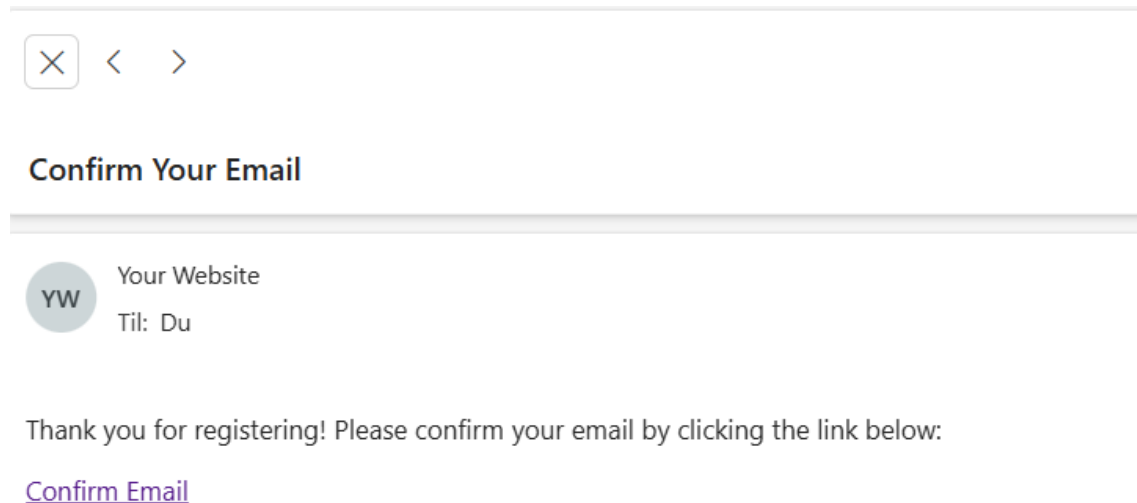


Figure 21: Confirmation e-mail sent with Gmail

9.4 Central Organization

"Mine lokallag"

A central organization user should be able to create local groups and assign an admin to each. To implement this functionality, and fulfill the requirement LGM1, key services, such as *localGroupService* and *localGroupAdminService*, were developed. The figure below illustrates the flow of messages/function calls between services in order to add an admin.

A local group admin must be a registered private user. In addition, it should not be possible to add a current admin as a new admin (as this is redundant). To ensure that these states are true, we used custom data annotations.

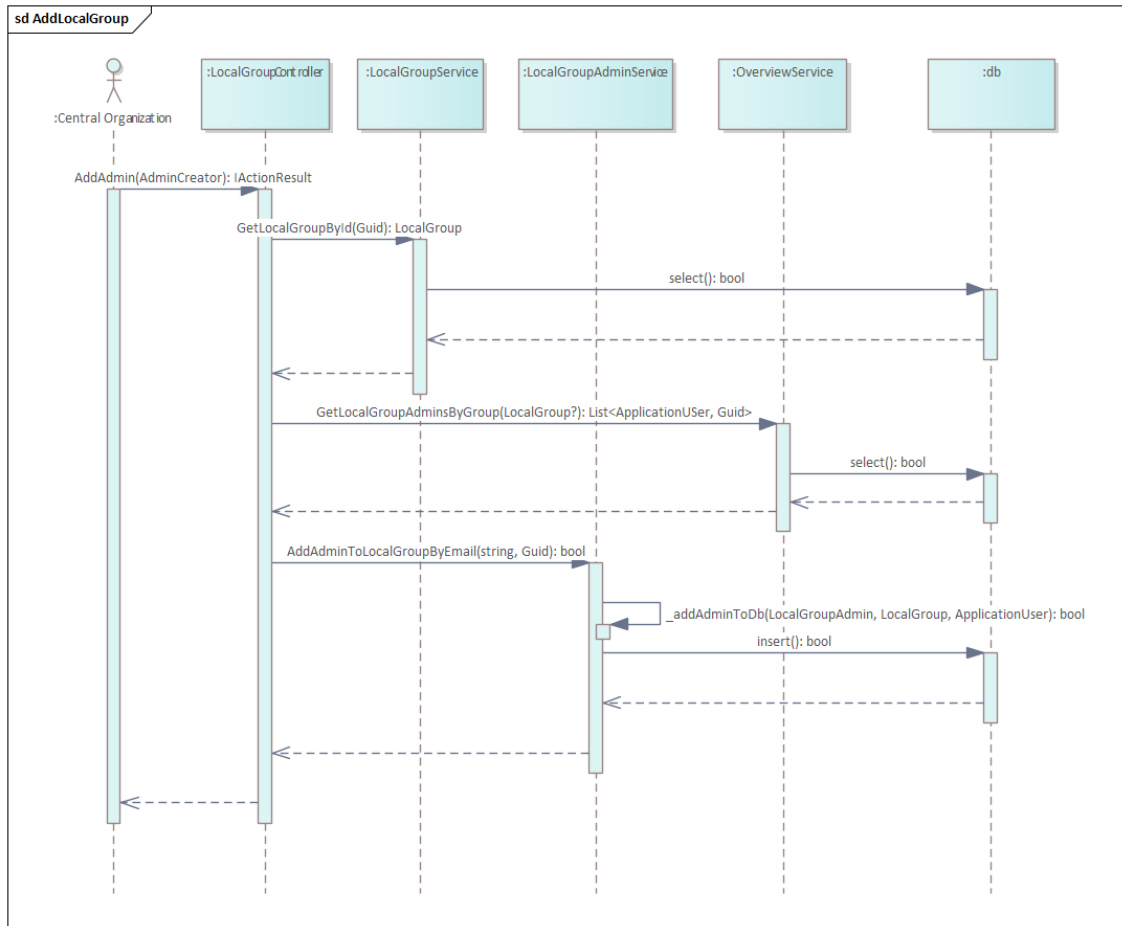


Figure 22: Interaction diagram for adding an admin to a local group

A private user can be admin for many local groups, and a local group can have many admins. An entity called *LocalGroupAdmin*, an "association class" [19] was created to enable this many-to-many relationship.

```

1      [PrimaryKey(nameof(LocalGroupId), nameof(UserId))]
2      public class LocalGroupAdmin
3      {
4          public Guid LocalGroupId { get; set; }
5          [ForeignKey(nameof(LocalGroupId))]
6          public LocalGroup LocalGroup { get; set; } = null!;
7          public string UserId { get; set; } = string.Empty;
8          [ForeignKey(nameof(UserId))]
9          public ApplicationUser User { get; set; } = null!;
10     }

```

Listing 5: LocalGroupAdmin entity defining relationships between local groups and users

Regarding the creation of new local groups, custom data annotations was also used to validate the correctness of a Norwegian postcode which should only contain numbers. The following code is taken from the *PostcodeFormatNumbersValidation* class:

```

1      // Code hidden for succinctness
2      {
3          if (value != null && value is string postcode)
4          {foreach (var c in postcode)
5              {if (!char.IsDigit(c))
6                  {return new ValidationResult(PostcodeValidationMessage);
7                  }
8              }
9              return ValidationResult.Success;
10         }
11         return new ValidationResult(PostcodeValidationMessage);
12     }

```

Listing 6: Validation logic for ensuring a postcode contains only digits

This validation can also be used in other models that require the same check.

9.5 Private User

"Profil"

A key feature prescribed by requirement UF7 is that a private user should be able to update personal details. The relationship between an *ApplicationUser* and *PrivateUser* is as displayed in the figure below:

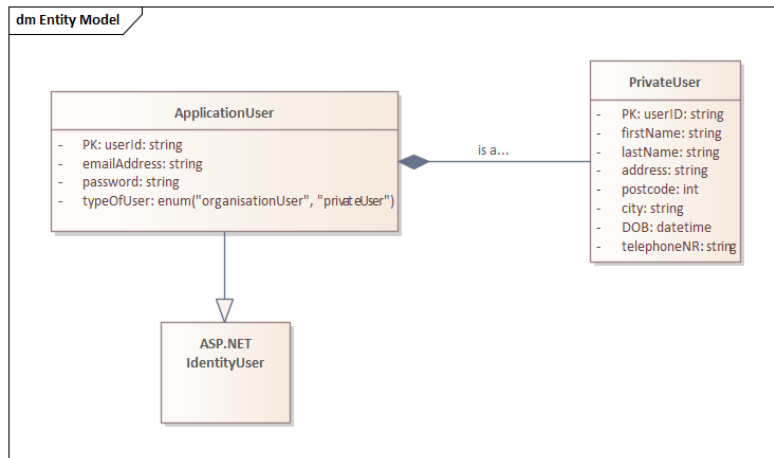


Figure 23: Relationship between ApplicationUser and PrivateUser

It was desired that the private user simultaneously updates the *ApplicationUser*, *PrivateUser*, and the person marked as the primary person associated with the private user. This was implemented by creating a view model called *Application-PersonalUser*. This approach allowed for updating the private user and application user simultaneously. In addition to this, when these models were updated, the primary person linked to the private user was identified and updated with the same personal details. The interaction diagram below illustrates how this functionality was implemented.

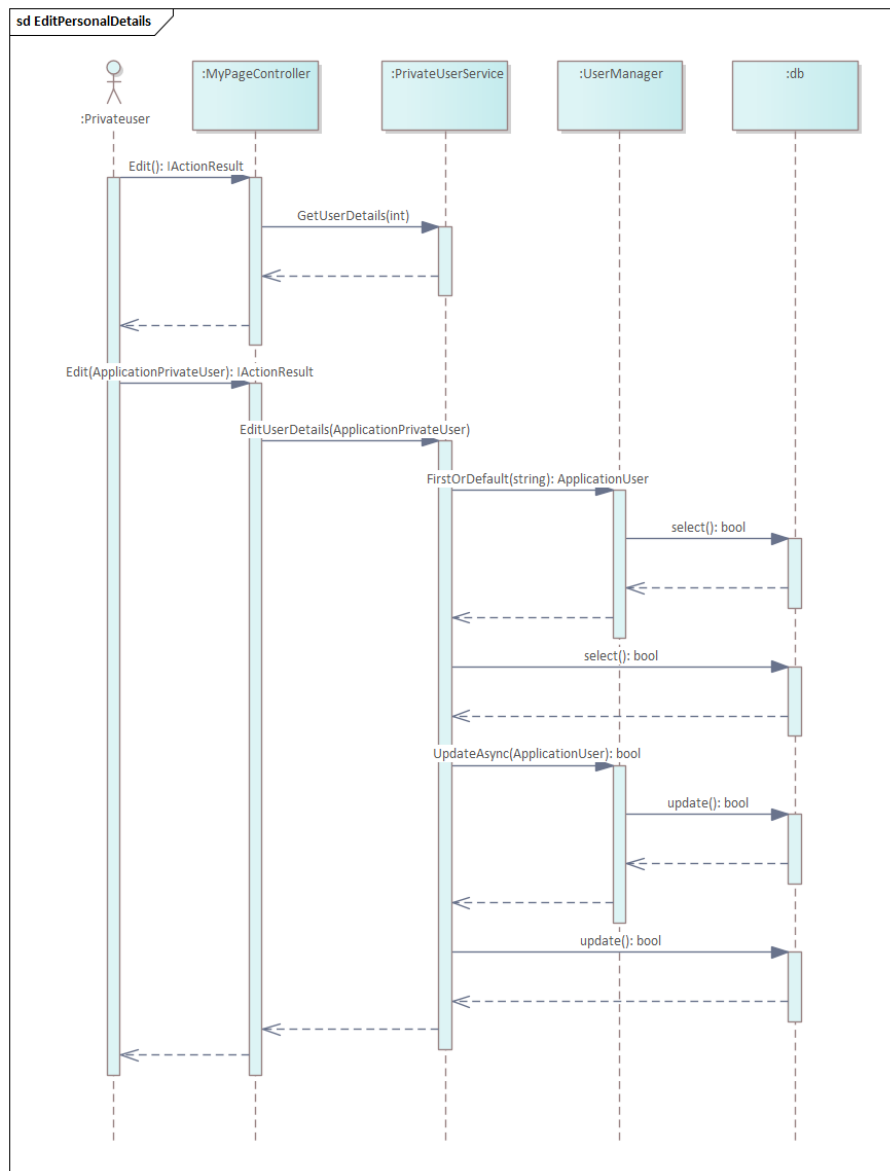


Figure 24: Interaction diagram for searching local group

"Mine brukere"

Legg til Bruker A user should be able to add persons that they can manage.

In order to fulfill the requirements MM11 and MM12, we needed a way of creating persons so that they could be connected with multiple private users. Consequently, we implemented an association class [19] as depicted below:

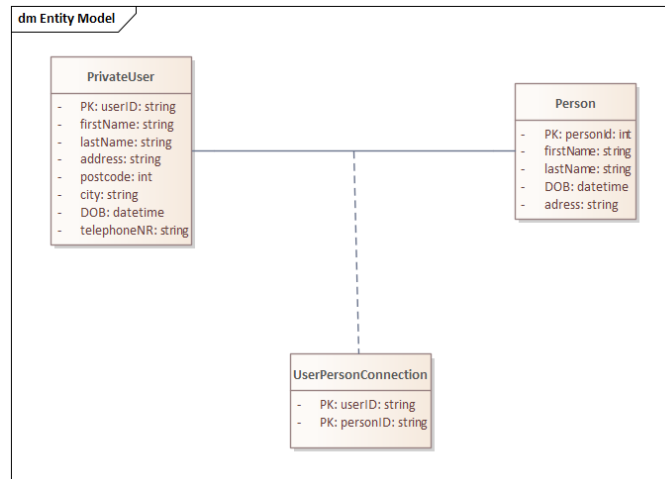


Figure 25: Relationship between a *PrivateUser* and *Person*

With the primary keys of *PrivateUser* and *Person* being used as the composite keys of a *UserPersonConnection*, each private user can be connected to every person once, and each person can be connected to every private user once. When a private user adds a person, then an *UserPersonConnection* is also created to reflect the relationship between the initiating private user and the created person.

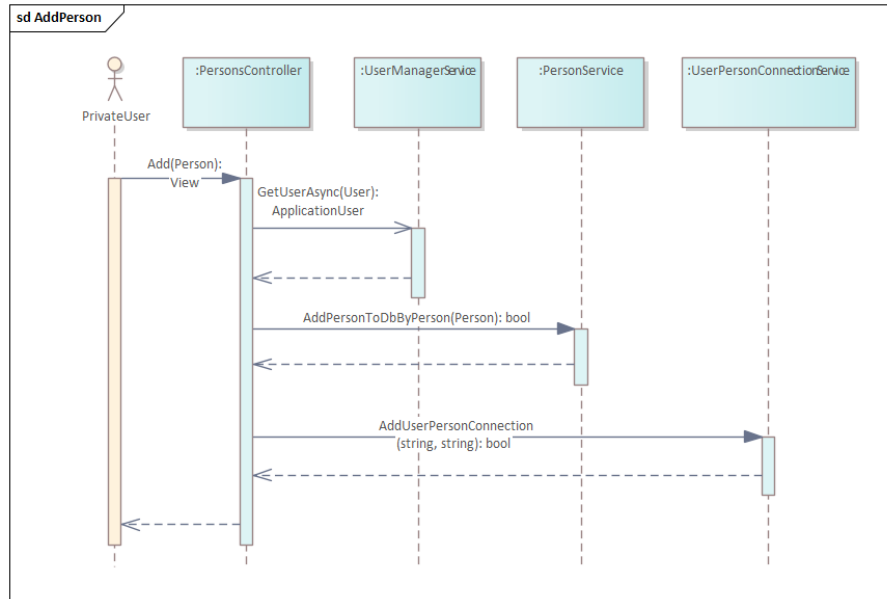


Figure 26: Sequence diagram that represents the calls between services to add a person. Database is not included (accessed through services)

In connecting private users and persons this way, sharing, transferring and deletion of persons, as prescribed by requirements MM12 and MM13, are respectively achieved by adding, adding & removing or simply removing a *UserPersonConnection* record.

PrivateUserId	PersonId
1 bcd8b433-a9cb-49bd-b11d-c0e7e43c4597	bcd8b433-a9cb-49bd-b11d-c0e7e43c4597
2 05ffba6a-810f-4914-b740-cbafc24106dd	05ffba6a-810f-4914-b740-cbafc24106dd
3 14c6aa47-6842-4f81-993a-1393e82e6493	14c6aa47-6842-4f81-993a-1393e82e6493
4 05ffba6a-810f-4914-b740-cbafc24106dd	68eb3d0a-2279-46c0-a77d-cf845684f0a5
5 05ffba6a-810f-4914-b740-cbafc24106dd	69e755fd-4943-48f6-8ecc-628277524b52
6 14c6aa47-6842-4f81-993a-1393e82e6493	0e5ca6ef-6479-47a5-9561-1f3e73a9f4aa
7 bcd8b433-a9cb-49bd-b11d-c0e7e43c4597	05ffba6a-810f-4914-b740-cbafc24106dd

Figure 27: UserPersonConnection table showing connections between private users and persons

Share persons

A private user can share a person connected to them with another user by entering the receivers email. The user accesses this functionality through pressing the button "Del" as seen in figure 12.

We use Bootstrap modals [17] to provide a pop-up dialog in which the user can

enter the email address of the private user that the person will be shared with. Most interactions with a controller and action result in a new view being returned to the user. However, in this case, we used Javascript and AJAX to prevent this from happening. Instead, form data is sent using the *fetch* method as shown below:

```
1 // Code hidden for succinctness...
2 fetch("/PrivateUser/Persons/SharePerson", {
3     method: "POST",
4     body: dataFromForm,
5 })
6 .then(response => response.json())
7 // Code continues...
```

Listing 7: JavaScript fetch request for sharing a person in the system

The controller returns JSON data based on the success of the functions it calls. For example:

```
1 // Code hidden for succinctness
2 if (!_aus.IsUserPrivateUser(desiredEmail))
3 {
4     return Json(new { success = false, errorMessage =
5         "Ugyldig e-postadresse. Brukeren må være registrert som private user." });
6 }
7 // Code continues...
```

Listing 8: Validation to ensure the email belongs to a registered private user

HTML code is then updated based on the value of the success key.

```
1 // Code hidden for succinctness
2 const updateInfo = document.getElementById("update-info");
3 updateInfo.textContent = "";
4 // Code hidden for succinctness
5 if (data.success)
6 {
7     updateInfo.textContent = "Personen ble delt!";
8 }
9 // Code continues...
```

Listing 9: Code snippet used in updating HTML after user successfully shares a person

When the modal is closed, the page is refreshed so that an updated list of persons may be shown to the user.

Transfer Persons

Transferring a person, as stated before, is achieved by adding & removing a *User-PersonConnection* record. This is achieved using the same mechanisms as described for sharing a person (Javascript, AJAX in conjunction with the *PersonsController*).

However, although the mechanisms are the same, there is a difference in decision logic. To transfer a person, we found that a chain of decision should be implemented as follows:

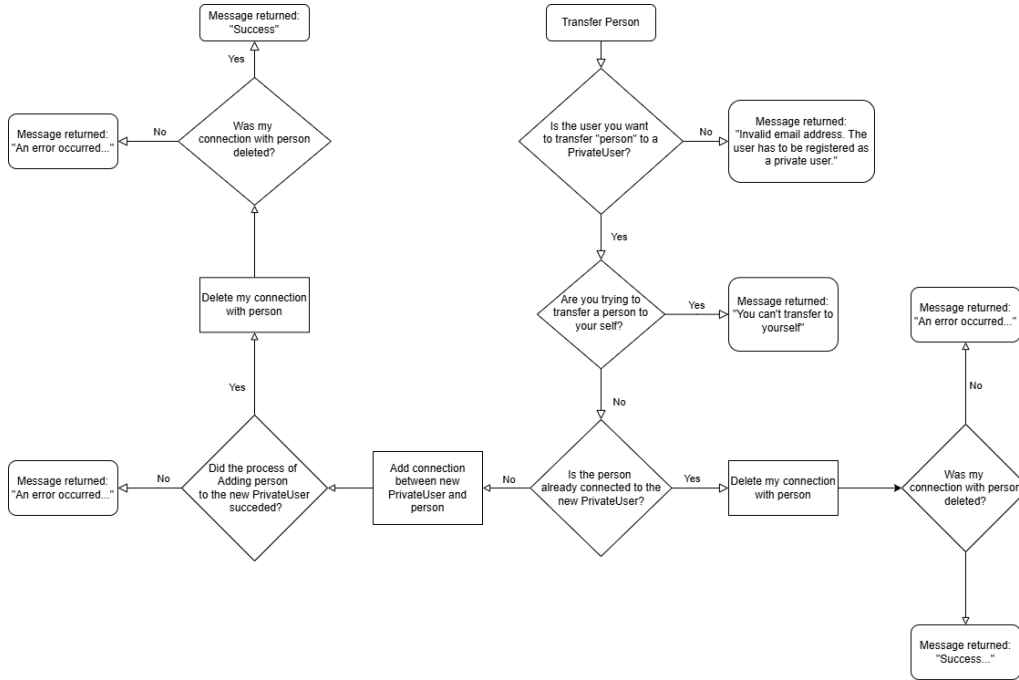


Figure 28: Flow diagram for implementation of transfer person

As this functionality was being implemented, it was observed that identifying the source of an error was not always straightforward. That is, as the controller uses multiple services to affect change in the model, using a *boolean* as the return value of functions does not tell us, the programmer, where an eventual error occurred. To meet the need for a more concrete understanding of where an error had occurred (for example, a function failing to add a *UserPersonConnection* to the database), we developed a simple class that combines boolean values with a message as shown in the code snippet below:

```
1 public class ResultOfOperation
2 {
3     public bool Result { get; set; }
4     public string Message { get; set; } = string.Empty;
5 }
```

Listing 10: ResultOfOperation class to standardize operation outcomes

Below we give a simple example on how this custom type was used:

```
1 public ResultOfOperation? DeleteUserPersonConnection(string userId, string personId)
2 {
3     // Code hidden for succinctness
4
5     var result = new ResultOfOperation
6     {
7         Result = false,
8         Message = string.Empty,
9     };
10
11     // Code hidden for succinctness
12
13     result.Result = _deleteConnection(privateUser, person);
14     if (!result.Result)
15     {
16         result.Message = "Unable to delete connection";
17         return result;
18     }
19     else
20     {
21         result.Result = true;
22         return result;
23     }
24 }
```

Listing 11: Example of using the ResultOfOperation type in a method for deleting user-person connections

Removing connection

Similar to sharing and transferal of persons, we use a Bootstrap modal to provide extra information for the user to make a decision on whether they wish to remove a person or not:

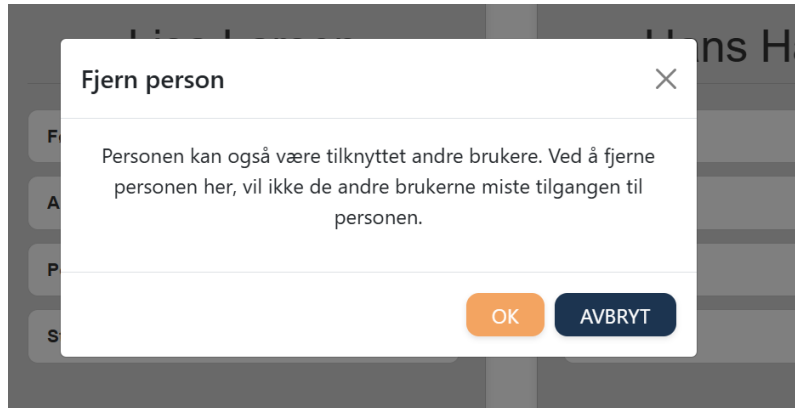


Figure 29: Modal with information presented to user before deleting user person connection

"Lokallag"

Become a member of a local group

In order to fulfill UF2 and UF3, the user needs to see a list of active local groups and choose an appropriate membership for one of their connected persons.



Figure 30: View of solution: Finding Available Local Groups

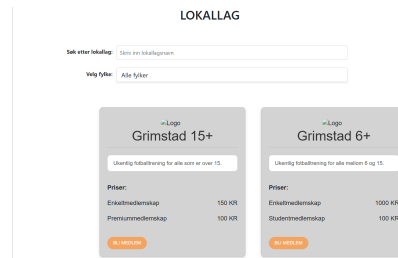


Figure 31: Overview of local groups in our implementation

In order for "Bli medlem" to work, we needed to use various models in connection with each other.

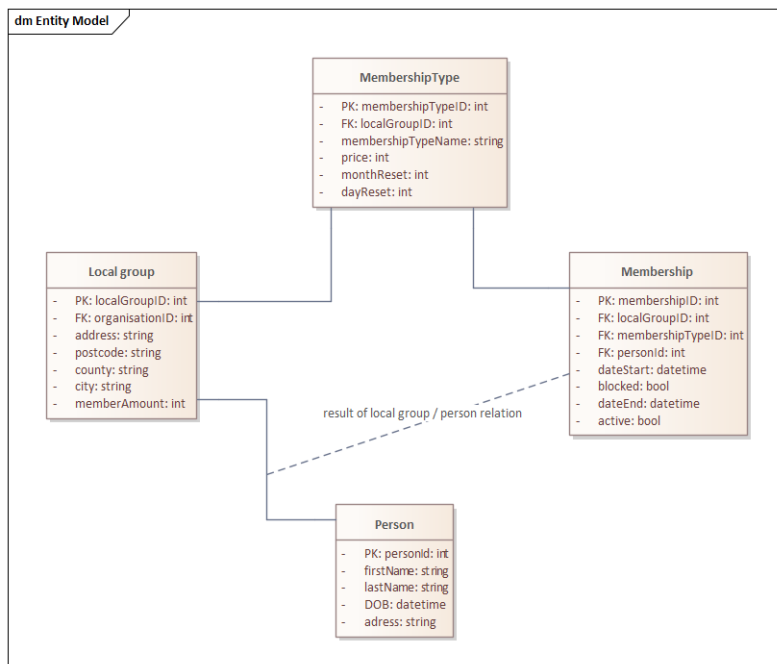


Figure 32: Relationships between Person, LocalGroup, Membership and MembershipType models

The above figure shows that every *Membership* is defined by the relationship between *Person* and *LocalGroup*. In addition, as local groups have different type of memberships, each *Membership* has an *MembershipType*.

```

1  [PrimaryKey(nameof(Id))]
2  public class Membership
3  {
4      public Guid Id { get; set; } = Guid.NewGuid();
5      public DateTime StartDate { get; set; }
6      public DateTime EndDate { get; set; }
7      public bool IsActive { get; set; }
8      public bool IsBlocked { get; set; }
9      public bool ToBeRenewed { get; set; }
10
11     [ForeignKey(nameof(MembershipTypeId))]
12     public Guid? MembershipTypeId { get; set; }
13     public MembershipType MembershipType { get; set; } = null!;
14
15     [ForeignKey(nameof(PersonId))]
16     public string? PersonId { get; set; }
17     public Person Person { get; set; } = null!;
18
19     [ForeignKey(nameof(LocalGroupId))]
20     public Guid? LocalGroupId { get; set; }
21     public LocalGroup LocalGroup { get; set; } = null!;
22
23     public ICollection<MembershipPayment> MembershipPayments { get; set; }
24     = new HashSet<MembershipPayment>();
25 }

```

Listing 12: Membership entity defining relationships and properties

The foreign keys represent the described relationships from figure 32. The *StartDate* is based on the timestamp for when a person became a member (using *DateTime.Now()*) and *EndDate* properties are based on the renewal date which every membership type must have when added by local group admins.

The code snippet below shows how the different properties in a *Membership* object are set. This function is called by the public function made available through the *IMemberService* interface.

```

1  private Membership? _addMembership(MembershipType mt, Person p, LocalGroup lg)
2  {
3      try
4      {
5          var currentMonth = DateTime.Now.Month;
6          var currentDay = DateTime.Now.Day;
7          var currentYear = DateTime.Now.Year;
8          var yearForReset = currentYear;
9
10         if (mt.MonthReset < currentMonth)
11         {
12             yearForReset = currentYear + 1;
13         }
14         else if (mt.MonthReset == currentMonth)
15         {
16             if (mt.DayReset <= currentDay)
17             {
18                 yearForReset = currentYear + 1;
19             }
20         }
21         var newMembership = new Membership
22         {
23             StartDate = DateTime.Now,
24             EndDate = new DateTime(yearForReset, mt.MonthReset,
25             mt.DayReset, 23, 59, 59, DateTimeKind.Utc),
26             IsActive = false,
27             IsBlocked = false,
28             ToBeRenewed = true,
29         };
30
31         newMembership.MembershipTypeId = mt.Id;
32         newMembership.LocalGroupId = lg.Id;
33         newMembership.PersonId = p.Id;
34
35         _db.Memberships.Add(newMembership);
36         //Code continues ...
37     }
38
39     public ResultOfOperation AddMembershipPaymentToDatabase
40     (Guid membershipId, Guid paymentId)
41     {
42         // Code hidden for succinctness
43
44         if (_addMembershipPayment(membership, payment))
45         {
46             resultOfOperation.Result = true;
47             return resultOfOperation;
48         }
49         // Code continues...
50     }

```

Listing 13: Code displaying the private function addMembership

Similar to the functionality for sharing, transferring and deletion of persons, the necessary controller is accessed through AJAX. A success result redirects the user to the payment gateway so that they can pay for the membership. When the payment is fulfilled, the membership is set as active.

```
1      // Code hidden for succinctness
2
3      if (result.success && result.redirectUrl)
4      {
5          console.log("Redirecting to:", result.redirectUrl);
6
7      // Code continues ...
```

Listing 14: Code displaying redirecting through AJAX

"Mine lokallag"

In order to fulfill AF6, a user should be able to see the local groups they are admin for (accessing them as an admin), and, in addition, see the local groups in which their connected persons are members (accessing them as a normal user). Due to the fact that we wish to present group name and status for admin groups, and group name for groups connected to active memberships, we use three "overview" view models [5], the one being composed of the two:

```
1      public class CompleteLocalGroupOverview
2      {
3          public List<AdminLocalGroupOverview>? AdminOverview { get; set; }
4          public List<PersonLocalGroupOverview>? PersonOverview {get; set;}
5      }
```

Listing 15: Code displaying the view models used for local group overview

AdminLocalGroupOverview and *PersonLocalGroupOverview* contains properties needed to present the following view:



Figure 33: View of the implemented "Mine lokallag"

Each overview is then passed to its respective partial view. Thus, the final view for "Mine Lokallag" consists of two partial views:

```

1      <div class="main-content">
2          <partial name="_AdminLocalGroupsOverview" model="@Model.AdminOverview"/>
3          <partial name="_MyPersonsLocalGroupsOverview" model="@Model.PersonOverview"/>
4      </div>

```

Listing 16: Code snippet of view for "Mine lokallag"

As shown in the entity model in chapter 7, information about the groups that a private user's persons are members of does not lie in one model. This information is found in how models are connected to one another. Rather than placing the logic to get this information expressly in the controller, we use the *OverviewService* accessed through its interface service as follows:

```

1      var overview = new CompleteLocalGroupOverview
2      {
3          AdminOverview = _os.GetAdminLocalGroupOverview(user.Id),
4          PersonOverview = _os.GetPersonLocalGroupOverview(user.Id),
5      };

```

Listing 17: Code snippet of using a service (*_os*) to generate the *PersonOverview*

A description of the logic found in *_os.GetPersonLocalGroupOverview(user.Id)* is presented in a flow diagram in Appendix N, as well as via the following link:

Click here to view the flow diagram.

(Note to ** in the diagram: These connections are fetched using eager loading [21]).

To access a specific local group, we make each group box a link to the corresponding group page as follows:

```
1 <a asp-area="PrivateUser"
2   asp-controller="MyLocalGroups"
3   asp-action="LocalGroupOverview"
4   asp-route-groupId="@group.LocalGroupId" class="localGroupMiniBox">
5   <div>
6     <p>@group.LocalGroupName</p>
7   </div>
8 </a>
```

Listing 18: Code snippet HTML code used so that a user can click and enter group page

"Lokallaginfo"

An admin can activate or deactivate a local group. This functionality was implemented to allow an organization user to create a local group and assign a private user as its admin. A local group remains inactive until an admin role is assigned. Once assigned, the admin can activate the group, making it visible to users. This implementation helped fulfill requirement LGM1 to allow a central organization user to create local groups with an assigned admin. The functionality was implemented by adding the property *Active* to the *LocalGroup* model, with a boolean type set to false by default. When a private user was granted admin rights for the local group, the admin could update the status to active through the view shown in the figure below.



Figure 34: Set local group to active

The buttons in the figure above were implemented to fulfill requirement AF6: As a private user, I want to manage multiple local groups with admin permissions to

oversee activities and members efficiently. This enables admins to easily navigate and manage their local groups. The buttons were implemented using the Anchor Tag Helper, as shown in the code below.

```
1 <a class="custom-orange-btn" asp-area="PrivateUser"
2   asp-controller="MyLocalGroups" asp-action="AdminGroupOverview"
3   asp-route-groupId="@Model.Id">Hjem</a>
```

Listing 19: Code displaying how buttons were implemented using Anchor Tag Helper

Part of managing a local group as an admin involved creating, deleting, and editing membership types for the group. To support this, the *MembershipType* model was created.

A *Membership* is connected to a *LocalGroup* and a *MembershipType*, and several *MembershipTypes* can be connected to a *LocalGroup*. The admin can define the name, price, and renewal date, which are the properties of the *MembershipType* model.

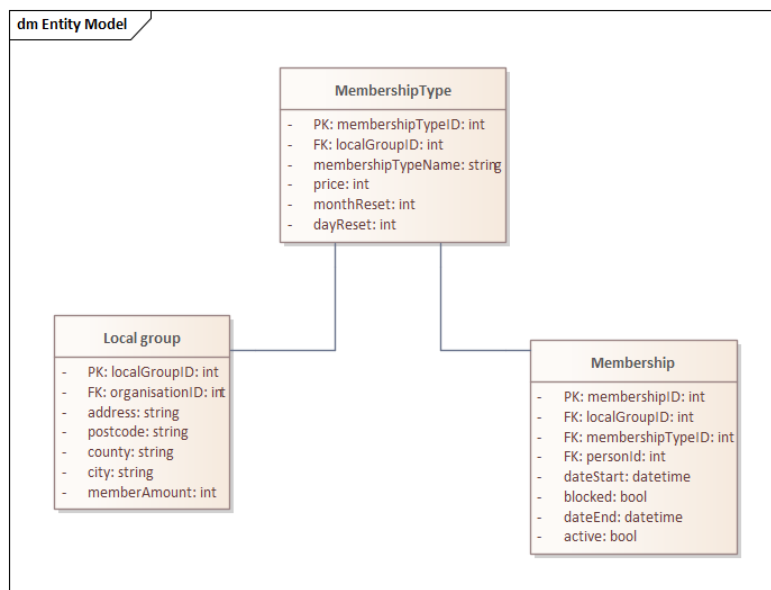


Figure 35: Entity model, Membership type

When a membership was created, a form was submitted to *MyLocalGroupController* containing the *localGroupId* and input from the admin. The controller then called the *IMembershipType* interface, implemented by *MembershipTypeService*, which added the new membership to the database with the *localGrouId* as

foreign key, and updated the list of memberships in the local group's model.

The attribute *monthReset* is of type int, but when the admin sets its value while creating a new membership type, it is displayed as text. This issue was resolved using JavaScript, as shown in the code below.

```
1 function MonthNumberToWords(number) {
2     const months = [
3         null, // Placeholder for index 0 (months start from 1)
4         "januar", "februar", "mars", "april", "mai", "juni",
5         "juli", "august", "september", "october", "november", "desember"
6     ];
7     return months[number] || "Invalid month";
8 }
```

Listing 20: Code to display converting numbers into month names.

The numbers for each month are displayed in a dropdown, but the JavaScript code converts the numbers into month names.

When an admin edits a membership type, a pop-up box is displayed, similar to when creating a new membership type. However, it was not desired to allow editing of the renewal date after it was set. This was resolved by making the input fields read-only. This is shown in the figure and code below.

```
1 <div class="form-group">
2     <label for="editMembershipMonth">Måned</label>
3     <input type="text" class="form-control" name="MonthReset" value=""
4         id="editMembershipMonth" readonly/>
5 </div>
```

Listing 21: Code to display setting input as read only.

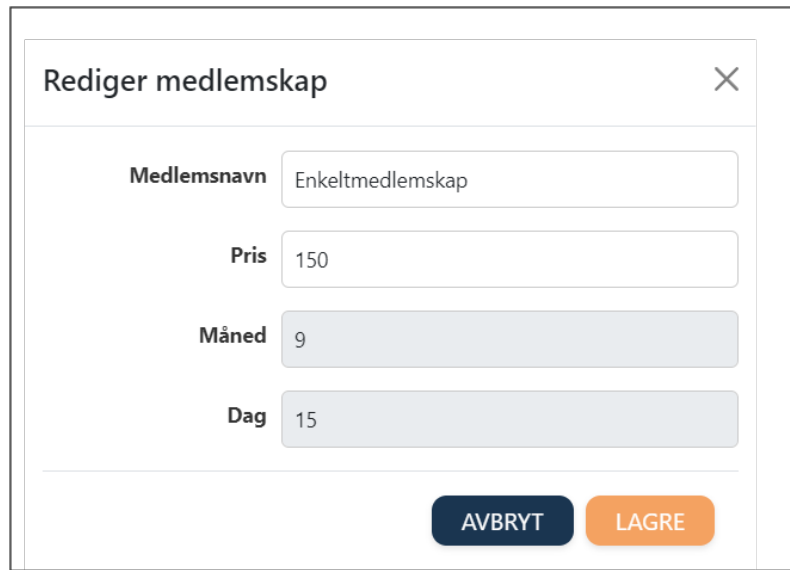


Figure 36: Editing membership type

"Medlemsliste"

To fulfill requirement MM8, an admin should be able to export a record of members. This functionality was implemented in the *AdminGroupMembers.cshtml* view, where the admin can see an overview of all members in the local group they manage.

When the admin clicks the export button, a JavaScript call is triggered to the *MyLocalGroupsController*. The controller first checks if there are active members in the group by calling a newly created interface implemented by the *GenerateMembershipListService* class, which queries the database. If active members are found, the controller calls the service again to generate the membership list and return it as a CSV file to the admin.

If the list is empty, a dialog box appears, asking the admin whether they want to generate the list anyway. If the admin selects "No," they are returned to the *AdminGroupMembers.cshtml* view without generating a list. If the admin selects "Yes," a call is made to the controller, which uses the *GenerateMembershipListService* to generate the list even if no members are present.

This process is illustrated in the flow diagram and interaction diagram below.

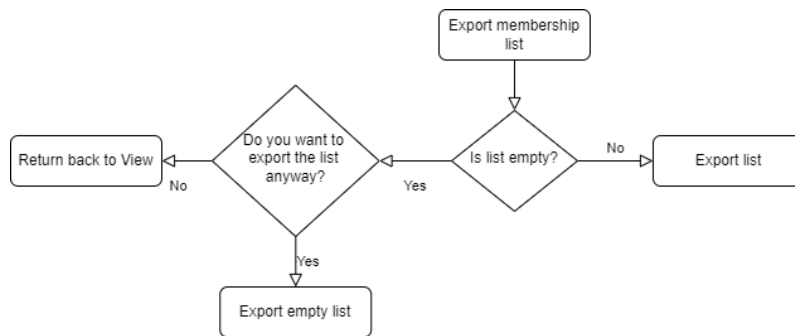


Figure 37: Flow diagram of how to export membership list

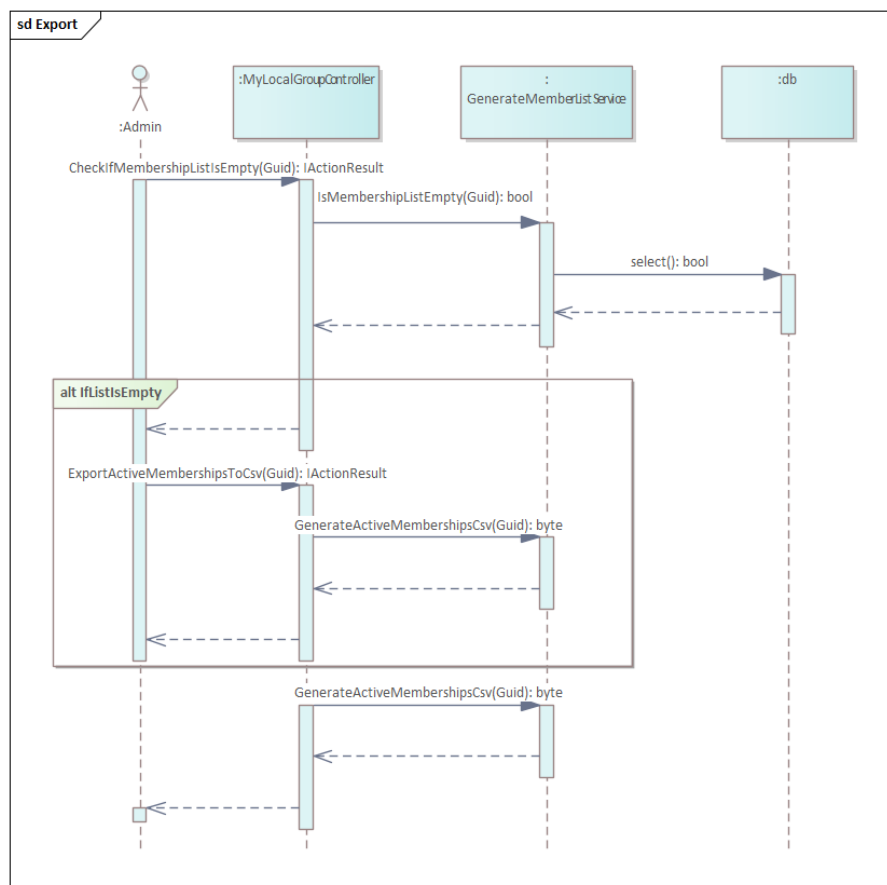


Figure 38: Interaction diagram of how to export membership list

In the same view, *AdminGroupMembers.cshtml*, the admin can search for members

in the list. This functionality is part of how requirement AF6 was addressed: As a private user, I want to oversee activities and members efficiently.

The view displays a table populated with the memberships in the local group. JavaScript has been used to add an eventListener to a search field above the table. This enables the table content to dynamically update based on what the admin types into the search field.

JavaScript achieves this by comparing the input in the search field with the content in the table column containing names. If the content matches the input, the corresponding row is displayed. If not, the row is hidden. The functionality is demonstrated in the code below.

```
1 if (matchesQuery && matchesActive && matchesMembershipType) {  
2     row.style.display = "";  
3 } else {  
4     row.style.display = "none";  
5 }
```

Listing 22: Code displaying how rows are sorted in JavaScript

The figure below shows how the *AdminGroupMembers.cshtml* was designed and implemented.

Grimstad 15+

HJEM

LOKALLAGINFO

MEDLEMSLISTE

Søk etter medlem:

Søk etter navn...

☐ Aktiv

☐ Ikke aktiv

Medlemskapstype:

Alle

Årstall:

Velg

Eksporter medlemsliste

Velg	ID	Navn	Alder	Adresse	Medlemskapstype	Medlem siden	Status	Blokkert
<input type="checkbox"/>	7d807a42-2d09-46b1-8e32-2182ae4c8297	Hans Hansen	30	Snarveien 17B, Grimstad	Enkeltmedlemskap	13.12.2024	Ikke aktiv	Nei
<input type="checkbox"/>	8dbb5407-f2ef-41e3-894b-560c0a142660	Lisa Larsen	32	Holvikaholsen 45, Grimstad	Premiummedlemskap	13.12.2024	Aktiv	Nei

BLOKKER MEDLEM

OPPHEV BLOKKERING

Figure 39: View of solution: AdminGroupMembers.cshtml

The requirements MM4 and MM19, which involve blocking and unblocking a member, are also implemented in this view. A member can be selected, and the "Block" or "Unblock" button can be clicked. Using JavaScript, a call is made to the *MyLocalGroupController*, which in turn calls the *MembershipService*. The *MembershipService* updates the database by setting the *isBlocked* attribute in the *Membership* model to true or false, based on the *membershipId* selected in the table. The status in the table is then updated based on whether the *isBlocked* attribute is true or false.

For now, the blocking functionality does not include any additional features. However, it was implemented with future system development in mind. The intended plan is that if a membership is blocked, the user associated with that membership will not be able to engage in activities within the local group. These functionalities, however, have not yet been implemented.

"Betalingen"

The payment page for private users allows them to have an overview of their payment history linked to the members associated with their accounts. This covers UF6, enabling a private user to view their payment history. This is implemented by creating a *PaymentViewModel* that is populated by the *PaymentController*.

```

1      var payments = await _db.MembershipPayments
2          .Include(mp => mp.Payment)
3          .Include(mp => mp.Membership)
4          .ThenInclude(m => m.MembershipType)
5          .Include(mp => mp.Membership.Person)
6          .ThenInclude(p=>p.UserPersonConnections)
7          .Where(mp => mp.Membership.Person.UserPersonConnections
8              .Any(upc=>upc.PrivateUser.Id == currentUser.Id))
9          .Select(mp => new PaymentListViewModel
10         {
11             //Logic to populate properties
12         })

```

Listing 23: Populating the PaymentViewModel from PaymentController

This query starts from the *MembershipPayments* table, which links payments to memberships. This way, we can access *Person* details, as well as *UserPersonConnections*. By doing so, we ensure that only the appropriate private users has access to their member's payment history.

In addition to viewing past payments, users can also view any outstanding payments, further addressing UF6. Each unpaid invoice is displayed with details such as the amount and associated membership. A button is provided for each unpaid invoice,

allowing users to proceed to the payment gateway to complete the transaction, completing UF4, that a user can complete their outstanding payments.

```
1      @if (!payment.Paid)
2      {
3          <a class="custom-orange-btn"
4              asp-area="PrivateUser"
5              asp-controller="Payment"
6              asp-action="Checkout"
7              asp-route-membershipId="@payment.MembershipId"
8              asp-route-paymentId="@payment.PaymentId">
9              Betal
10             </a>
11     }
```

Listing 24: Code to display a pay button

Upon successful payment, the system updates the payment status of the *Payment* entry to *true*, which is reflected immediately in the user’s view. For shared members, this update is synchronized across all connected users, ensuring consistency and transparency.

Betalinger					
Medlem	Medlemskap	Pris	Status	Betalt dato	Gyldighetsperiode
Hans Hansen	Enkeltmedlemskap	150 NOK	Betalt	13.12.2024	13.12.2024-15.09.2025
Lisa Larsen	Studentmedlemskap	100 NOK	ikke betalt	-	13.12.2024-11.12.2025

Figure 40: Outstanding Payment

Betalinger					
Medlem	Medlemskap	Pris	Status	Betalt dato	Gyldighetsperiode
Hans Hansen	Enkeltmedlemskap	150 NOK	Betalt	13.12.2024	13.12.2024-15.09.2025
Lisa Larsen	Studentmedlemskap	100 NOK	Betalt	13.12.2024	13.12.2024-11.12.2025

Figure 41: Payment Processed

Due to time constraints, the functionality to cover UF5, that a user can download receipts for payments, and part of UF4, that a user can pay for outstanding membership fees was not implemented. A "Download Receipt" button is intended for completed payments to allow users to generate a PDF receipt containing payment details. We are lacking the back-end, as well as the front-end, functionality to support this feature. For the payment of outstanding membership fees, we have not yet implemented a "generate invoice" functionality for admins to renew the memberships each year, meaning there is a lot of work yet to be done in that area.

10 Testing and Validation

This section summarizes the testing process and results. Tests were developed based on user stories, with each user story assigned its own test case. Each test case included predefined steps to verify if the system behaved as expected. Test results were recorded as either "passed" or "failed" based on whether the system fulfilled the expected behavior.

Figure 42 presents the results of all test cases. Detailed descriptions and execution steps for each test case are provided in Appendix M, as well as directly here: *Test Report*.

Name: VIO					
Project ID: 22					
Report Date: 12.12.2024					

SUMMARY		
Total Test Cases		17
Executed		16
Pass		14
Fail		2
Not Executed		1

FUNCTIONAL TESTING					
Test Case ID	Use Case ID	Description	Pass/Fail/Not Executed	Test Date	Comment
TC01	UF1	As a user, I want to create an account in the system so that I can access the system's functionalities.	Passed	12.12.2024	
TC02	UF2	As a user, I want to see all available local groups, so that I can choose one that suits my needs.	Passed	06.12.2024	
TC03	UF3	As a user, I want to register a person connected to me in a local group, so that they can become a member and access group functionalities.	Passed	12.12.2024	
TC04	UF4	As a user, I want to pay my outstanding membership fees using an automated payment process so that I can complete my payment easily.	Passed	12.12.2024	
TC05	UF7	As a private user, I want to update my personal details so that my information is accurate.	Partially Passed	09.12.2024	Not giving a warning about where the invalid information is entered
TC07	MM8	As a local group admin, I want to export the record of members so that I can easily import it into the system used for end-of-year reporting.	Passed	12.12.2024	
TC09	MM11	As a user, I want to add and manage (edit/delete) multiple persons (e.g., partner, children) under my account so that I can administer their memberships, handle administrative tasks, and centrally manage all related activities.	Partially Passed	09.12.2024	Not giving a warning about where the invalid information is entered
TC10	MM12	As a user, I want to share a person with another user so that they also can manage (edit/delete) the persons memberships connected to me, allowing shared responsibilities (e.g., parents managing children's activities).	Passed	09.12.2024	
TC11	MM13	As a user, I want to transfer persons that I manage to another user so that they can manage them.	Passed	09.12.2024	
TC12	MM17	As a user, I want to see all the persons connected to my account, and their memberships, so that I am aware of which memberships I have.	Passed	12.12.2024	
TC13	MM18	As a user, I want to cancel memberships connected to my persons, so that they are no longer a part of the local group.	Passed	12.12.2024	
TC14	LGM1	As a central organization user, I want to create local groups with an assigned admin so that new communities can be formed.	Passed	06.12.2024	
TC15	UF5	As a private user, I want to download receipts for payments so that I can keep a record of my transactions.	Not Executed		
TC16	UF6	As a private user, I want to see details of previous payments or outstanding payments so that I can stay informed about my financial obligations.	Passed	12.12.2024	
TC17	AF6	As a private user, I want to manage multiple local groups once admin permissions have been granted so that I can oversee activities and members across different groups efficiently.	Passed	12.12.2024	
TC18	MM4	As a local group admin, I want to be able to block members from being able to engage in local group activity so that the local group is protected from unwanted behavior.	Passed	12.12.2024	
TC19	MM19	As a local group admin, I want to unblock a person's membership so that they can once again participate in the local group's activities.	Passed	12.12.2024	

Figure 42: All test cases

11 Discussion

11.1 Process

When this project began, much of the process focused on planning and research. A significant amount of time was spent creating entity diagrams, use cases, and use case descriptions. Collaboration within the group worked well, with tasks being evenly distributed among the members. Over time, everyone in the group gained substantial new knowledge in terms of software architecture and development skills.

In hindsight, it could be argued that too much time was spent on processes like developing user stories and use case descriptions. As the team's knowledge increased over the weeks, and also due to scope creep, the initial use case descriptions were later viewed as nearly useless, as they no longer aligned with the product being developed. Looking back, it might have been better to approach the planning process as an even more iterative one, considering the low initial experience and knowledge level of most group members. On the other hand, the well-structured planning proved valuable as a solid foundation as the project progressed.

The team's use of Git has been effective, with one member acting as the Git master and another managing Jira. Overall, this process has run smoothly. However, one key takeaway is the importance of remaining mindful of potential bottlenecks during development. This is an insight gained through hands-on experience, although we cannot say we know exactly how to prevent bottlenecks completely.

Key functionalities often needed to be completed before other tasks could proceed, leading to occasional delays as some team members had to wait for others to finish. Despite this, the team adapted well by utilizing these periods to focus on report writing, which was always an essential parallel task for the project.

11.2 Product

An entity model was developed early on to lay the foundation for how the system would be built. Initially, there was a plan and a goal to create a system where local groups could have well-functioning accounts, allowing for the creation of events with registrations and payment fees, as well as a messaging system where members could communicate with each other and the local group administrators. Over time, these goals had to be scaled back due to constraints, particularly with respect to time and the team's level of knowledge at the time.

On one hand, it can be argued that the product has not fully met all the initially specified requirements. However, the team prioritized the requirements during the development process, focusing only on the highest-priority features. In light of this, the product satisfies many of the requirements, such as making an account, creating

local groups, getting admin rights, add, share and transfer persons, and become a member of a local group through a payment service. However, it falls short in certain areas. An example of this is that, in retrospect, it became clear that if the system allowed a local group admin to generate invoices, there should also be a way for them to view the invoices generated or receive confirmation that they were sent out. As the system stands now, admins cannot be certain that the invoices were successfully created and distributed to the intended members. This need was identified during development but has not yet been implemented, leaving room for further development.

Similarly, the functionality for sharing a person associated with a private user has some limitations. A restriction was put in place to allow a person to be shared by a maximum of two users, ensuring better control over who has access to the person's details. There is also a feature for transferring a person from one user to another. However, it was discovered during development that, as the system is currently designed, a person can still be transferred further to other users, even if they are already shared with two users. This could lead to a lack of clarity about who has access to the person's data, presenting another area for potential improvement and future development.

11.3 Implementation

Looking back at the implementation, several aspects worked well. Due to solid planning, there was clarity about what needed to be implemented. There were, however, gaps in knowledge about how to implement our plans effectively. For example, the use of service interfaces had to be studied and understood, as it was unfamiliar to the team. Over time, the team improved in breaking down functionality into concise and focused interfaces and services, adhering to the Single Responsibility Principle and the Interface Segregation Principle. This made the code more readable, understandable, and easier to maintain and extend.

Methods like AJAX were also employed. Once the concept was grasped, it worked effectively, especially in the implementation of the registration page.

Despite thorough planning, several adjustments had to be made during the project. For example, the logic involving Person, Private User, and Shared Person needed changes. Initially, the entity model defined a one-to-many relationship between Private User and Person. A Shared Persons table was used to record the persons that were shared between private users. As the system evolved, this approach caused issues. For instance, when a private user attempted to remove a shared person it had created, the person was entirely deleted from the system. This meant that the other private user also lost access to this person, an outcome which was clear undesirable.

The issue was resolved by modifying how persons were created and managed as explained in chapter 9. This implementation worked well and successfully addressed the requirement to remove a user's association with a person without entirely deleting the person if they were still connected to another user.

In several parts of the implementation, JavaScript is required for functionality. This implementation works well and provides users with a smoother experience when interacting with the web application. However, it does present challenges for users who do not have JavaScript enabled. This limitation was not addressed in the initial implementation but has been identified as an area for potential future development. A possible solution could be to provide a fallback link that redirects users to a page where they can manually fill out the form, bypassing the JavaScript dependency. Another option would be to inform users that this functionality requires JavaScript, ensuring they are aware of the limitation.

While these are potential future solutions, the current implementation does not meet the requirements for users without JavaScript, leaving room for improvement in addressing this issue.

11.4 Challenges

The project has presented several challenges and issues. Time has been a significant factor, preventing the completion of many planned implementations. In addition to time constraints, the team encountered challenges for which no ideal, functional, or desired solutions were found during this phase.

For instance, in the design and planning stages, it was envisioned that an admin could add multiple types of memberships to a local group. However, this turned out to be more problematic than initially anticipated. Questions surrounding membership renewal in relation to deleting, sharing, and transferring persons proved particularly challenging. Due to time constraints, it was decided not to address these issues in the current scope. However, it is recognized that this functionality would be valuable to add in the future.

A decision was made to focus initially on implementing single memberships and ensuring robust functionality around this core feature. This approach allowed the team to deliver a functional solution while laying the groundwork for more advanced membership types to be added later.

11.5 Methodological Reflection

The competitive analysis leading to the results in Chapter 5 was conducted using publicly available data from the websites of each platform. While this approach

provided valuable insights into pricing models, target audiences, and key features, it is acknowledged that certain details—such as real-world user feedback or advanced technical specifications—could not be fully explored. Future research could include interviews with users or testing of these platforms to gain deeper insights into their usability and performance.

11.6 For the future

Although many functionalities have been implemented during the project, several features that were initially planned were either not included or only partially implemented.

One such feature was the ability to transfer a person from a user to a non-existent user via email. The idea was that the recipient would receive an email with a link informing them that a person has been assigned to them. By clicking the link, they could create a new account on the site, with the assigned person becoming their primary person. This functionality was envisioned for cases like a child being administered by a parent, who could then transfer the administration to the child when they become independent. With the email service functionality now implemented, this feature seems achievable in the future.

It was also planned that a private user would receive an email notification whenever a person is shared or transferred. This functionality could also be added later without much difficulty.

As mentioned earlier, there is a need for admins to at least receive confirmation that invoices have been generated and sent. Ideally, there should also be a view showing which members have received them.

Another identified need is for a private user to have a view where they can see who else has access to the persons associated with them. This would ensure that a person cannot be transferred further without the private user being aware of it.

In the early stages of planning, it was a plan that admins could create events for their local group, allowing members to register by paying a fee. This feature had to be de-prioritized but is still a highly desired addition for future development. Similarly, several requirements involved allowing members and admins to send private messages to each other. This feature was not implemented due to time constraints, but with more time and experience, it could be added to the system in the future.

As previously mentioned, there was an effort to introduce multiple types of memberships for local groups, such as family memberships. Toward the end of the project, this was considered too complex and was unfortunately de-prioritized. Currently, admins can add a membership type and price, but it only supports single mem-

berships. In the future, with more time and experience, this could be expanded to include family memberships. However, this would require developing a robust structure for memberships and their renewal, which has not yet been addressed.

12 Conclusion

The development of a complete membership management system for local groups, as outlined in this report, was a more complex endeavor than initially anticipated. Significant effort was dedicated to the planning phase to ensure that the foundational architecture was robust enough to support the system’s scalability and adaptability. This emphasis on planning allowed us to focus on essential functionalities while leaving room for future expansion.

The implemented solution successfully addresses the core requirements, including membership management, payment handling, and role-based access. These achievements demonstrate the feasibility of creating a streamlined and centralized platform tailored to the needs of community organizations. While the process was marked by strong collaboration and effective use of development tools, some areas—particularly non-functional requirements like performance optimization and advanced security measures—remain underdeveloped due to time constraints.

Despite these limitations, the project yielded valuable insights into both technical and organizational aspects of software development. The system offers a solid foundation for future iterations, with potential for real-world application in simplifying administrative tasks for local groups. By providing a modular and user-friendly interface, the solution stands out compared to existing platforms, particularly for smaller organizations with limited resources.

Looking forward, future work should prioritize addressing identified gaps, such as enhancing security, implementing cross-platform compatibility, and refining the user experience based on feedback from real-world testing. Expanding the system’s functionality to include features like receipt downloads, automated membership renewal, and multi-member memberships would further increase its value to users.

In conclusion, the project validates the potential of a centralized, role-based system to contribute to membership management for local groups, laying the groundwork for continued development and broader adoption.

References

- [1] [Online; accessed 12. Dec. 2024]. Mar. 2006. URL: https://staff.cs.utu.fi/~jounsmmed/doos_06/material/DesignPrinciplesAndPatterns.pdf.
- [2] */Spond - Forsiden/*. [Online; accessed 29. Oct. 2024]. 2024. URL: <https://www.spond.com/no/>.
- [3] AndriySvyryd. *Creating and Configuring a Model - EF Core*. [Online; accessed 12. Dec. 2024]. Nov. 2024. URL: <https://learn.microsoft.com/en-us/ef/core/modeling>.
- [4] *ASP Ajax*. [Online; accessed 06. Dec. 2024]. URL: https://www.w3schools.com/asp/asp_ajax.asp.
- [5] Nate Barbettini. *Introduction · Little ASP.NET Core Book*. [Online; accessed 12. Dec. 2024]. Dec. 2024. URL: <https://nbarbettini.gitbooks.io/little-asp-net-core-book/content>.
- [6] *Brønnøysundregistrene - Forsiden*. [Online; accessed 29. Oct. 2024]. Oct. 2024. URL: <https://www.brreg.no>.
- [7] *Consio: About*. [Online; accessed 29. Oct. 2024]. Oct. 2024. URL: <https://www.consio.no/homepages/about-consio>.
- [8] *Consio medlemssystem | Funksjoner*. [Online; accessed 29. Oct. 2024]. Oct. 2024. URL: <https://www.consio.no/homepages/funksjonerogmoduler>.
- [9] *Consio medlemssystem | Priser*. [Online; accessed 29. Oct. 2024]. Oct. 2024. URL: <https://www.consio.no/homepages/priser>.
- [10] *Cornerstone*. [Online; accessed 29. Oct. 2024]. Oct. 2024. URL: <https://cornerstone.no>.
- [11] *Frifond – Vi støtter dere*. [Online; accessed 29. Oct. 2024]. Sept. 2024. URL: <https://www.frifond.no>.
- [12] GeeksforGeeks. “MVC Design Pattern”. In: *GeeksforGeeks* (Oct. 2024). URL: <https://www.geeksforgeeks.org/mvc-design-pattern>.
- [13] IEvangelist. *Dependency injection - .NET*. [Online; accessed 12. Dec. 2024]. Aug. 2024. URL: <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>.
- [14] Davidse Jenny. *API fundamentals*. IBM Developer. Dec. 15, 2020. URL: <https://developer.ibm.com/articles/api-fundamentals/#benefits-of-api-development4> (visited on 12/05/2024).
- [15] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. 8th. Pearson, 2021.
- [16] Peep Laja. *8 Web Design Principles & Laws That Work*. CXL. Sept. 26, 2022. URL: <https://cxl.com/blog/universal-web-design-principles/> (visited on 12/09/2024).
- [17] Mark Otto, Jacob Thornton, Bootstrap contributors. *Modal*. [Online; accessed 12. Dec. 2024]. May 2024. URL: <https://getbootstrap.com/docs/5.3/components/modal>.
- [18] Learn Microsoft. *Entity Framework Overview - ADO.NET*. Sept. 15, 2021. URL: <https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview> (visited on 12/05/2024).

- [19] *Rational Software Modeler 7.5.5*. [Online; accessed 12. Dec. 2024]. Mar. 2021. URL: <https://www.ibm.com/docs/en/rsm/7.5.0?topic=diagrams-association-classes>.
- [20] Rick-Anderson. *Introduction to Identity on ASP.NET Core*. Aug. 30, 2024. URL: <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-9.0> (visited on 12/05/2024).
- [21] roji. *Eager Loading of Related Data - EF Core*. [Online; accessed 13. Dec. 2024]. Feb. 2024. URL: <https://learn.microsoft.com/en-us/ef/core/querying/related-data/eager>.
- [22] Ravi Sethi. *Software engineering: basic principles and best practices*. Cambridge: Cambridge university press, 2023. ISBN: 978-1-316-51194-7.
- [23] SmartOrg | Sameie, lag og forening. Helt enkelt. [Online; accessed 29. Oct. 2024]. Oct. 2024. URL: <https://smartorg.no>.
- [24] Steve Smith. *Overview of ASP.NET Core MVC*. June 17, 2024. URL: <https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-9.0> (visited on 12/05/2024).
- [25] tdykstra. *Areas in ASP.NET Core*. [Online; accessed 12. Dec. 2024]. Nov. 2024. URL: <https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/areas?view=aspnetcore-9.0>.

A Product Vision

The complete Product Vision is available in the appendices folder here: *ProductVision.pdf*

B Group Contract

The Group Contract is available in the appendices folder here: *group-contract.pdf*

C Product Info

Product info can be found in the appendices, here:

Username and Password for each role: *product-info-username-and-password-information.pdf*

Initializing code: *product-info-InitialiserCode.cs*

D Time Sheets

The time sheets can be found for each person in the appendices folder here:

Nancy Rønqist Erichsen: *time-sheet-nancy-erichsen.pdf*

Adam Hazel: *time-sheet-adam-hazel.pdf*

Gunn Marita Jomås-Britten: *time-sheet-gunn-marita.pdf*

Nora Lior: *time-sheet-nora-lior.pdf*

E Meeting Minutes

The complete set of Meeting Minutes is available in the appendices folder here: *VIOMinutes.pdf*

F Demo Video

The Demo Video showcasing the product is available in the appendices folder here: *exam201-video1.mp4*

G Git Shortlog

The Git Shortlog is available in the appendices folder here:

git.log

H Sprint Reports

The complete Sprint Report can be found in the appendices folder here: *sprint-reports.pdf*

I Individual Reports

The individual reports can be found for each person in the appendices folder here:

Nancy Rønqist Erichsen: *individual-report-nancy-erichsen.pdf*

Adam Hazel: *individual-report-adam-hazel.pdf*

Gunn Marita Jomås-Britten: *individual-report-gunn-marita.pdf*

Nora Lior: *individual-report-nora-lior.pdf*

J Requirements

The complete Requirements is available in the appendices folder here: *Requirements.pdf*

K Figma Views

The complete Figma Views is available in the appendices folder here: *FigmaViews.pdf*

L Entity Model

The full render of the Entity Model is available in the appendices folder here: *entity-model.svg*

M Test Report

The complete Test Report is available in the appendices folder here: *TestingAnd-Validation.pdf*

N Flow Diagram for Get Person

The full scale model of the flow diagram for get personoverview is available in the appendices folder here: *GetPersonOverviewFlowDiagram.svg*