



Medlemskapssystem og Betalingsssystem

Prosjektrapport

Bachelorprosjekt i Informasjonsteknologi 2025

Gruppe 38

Ask Norli

Adrian Hoff

Simen Thams

Adina Heia

Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo

Besøksadresse: Holbergs plass, Oslo

Telefon: 22 45 32 00

PROSJEKT NR.

38

TILGJENGELIGHET

Åpen

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL	DATO
Medlemskapssystem og Betalingssystem	23.05.2025
	97 Sider
PROSJEKTDELTAKERE	INTERN VEILEDER
Ask Norli – s374959 Adina Heia – s374989 Adrian Hoff – s374946 Simen Thams – s374935	Terje Gjøsæter

OPPDRAKGIVER	KONTAKTPERSON
Elektronisk Forpost Norge	Tom Fredrik Blenning

SAMMENDRAG
Bachelorprosjektet er gjennomført av fire studenter ved OsloMet i samarbeid med Elektronisk Forpost Norge (EFN). Prosjektet omhandler videreutvikling av et påbegynt medlemskapssystemet for medlemsregistrering i EFN, samt utvikling av et åpent kildekode betalingssystem med støtte for videre utvikling og integrering i medlemskapssystemet etter bachelorperiodens slutt.

3 STIKKORD

Medlemskapssystem, Docker-miljø, Open source

Sammendrag

Dette bachelorprosjektet har gått ut på å videreutvikle og ferdigstille et påbegynt medlemskapssystem for EFN (Elektronisk Forpost Norge) samt utvikle et uavhengig betalingssystem. Prosjektgruppen har implementert alle nødvendige krav og funksjoner for å gjøre medlemskapssystemet produksjonsklart. Det nye betalingssystemet er utviklet med funksjoner som skal gjøre det mulig å betale med vanlig faktura og kontantfaktura. Betalingssystemet er utviklet som åpen kildekode for å sikre gjenbrukbarhet og åpenhet.

Begge systemene er grundig dokumentert for å sikre en smidig overtakelse og videre utvikling. Resultatet av oppgaven er et produksjonsklart medlemskapssystem med enklere registrering og betaling samt effektivisert medlemshåndtering. Betalingssystemet tilrettelegger for fleksible betalingsmetoder og gir brukere valgfrihet. Modulen er designet for å være fleksibel slik at den enkelt kan integreres av andre aktører og tilpasses ulike systemer.

Forord

Denne rapporten utgjør sluttrapporten for vår bacheloroppgave i informasjonsteknologi ved OsloMet, våren 2025. Dokumentet inneholder overordnet en gjennomgang av det tekniske aspektet, utviklingsprosessen og evaluering av prosjektet.

Samarbeidet med Elektronisk Forpost Norge (EFN) ble etablert gjennom en av gruppemedlemmene bekjentskap med daglig leder Tom Fredrik Blenning. Gjennom denne kontakten fikk vi innsikt i behovene deres, som dannet grunnlaget for prosjektet. EFN hadde et allerede fungerende medlemskapssystem for EFN der man kunne betale for å bli medlem i organisasjonen. Dette systemet ville de få ryddet opp og oppgradert.

Vi ønsker å rette en spesiell takk til vår veileder hos Elektronisk Forpost Norge (EFN), Tom Fredrik Blenning, for veiledning og konstruktive tilbakemeldinger gjennom prosjektet. Vi vil også takke Bjørn Remseth, nestleder i EFN, som med sin innsikt og erfaring fra det opprinnelige medlemskapssystemet har bidratt med verdifulle innspill underveis. Til slutt ønsker vi å takke vår skoleveileder, Terje Gjøsæter, for faglig støtte og veiledning gjennom hele prosjektperioden.

Innholdsfortegnelse

Innhold

Sammendrag.....	3
Forord	3
Innholdsfortegnelse	4
1. Innledning	7
1.1 Presentasjon av gruppen.....	7
1.2 Presentasjon av oppdragsgiver	8
1.3 Oppdrag	9
1.3.1 Bakgrunn for oppdraget - dagens situasjon	9
1.3.2 Oppdraget.....	10
1.3.3 Rammebetingelser og mål	10
1.3.4 Forventede gevinster	12
2. Prosessdokumentasjon	13
2.1 Innledning	14
2.2 Planlegging og metode	14
2.3 Teknologi og verktøy vi har brukt	15
2.3.1 Oppsett av prosessverktøy	21
2.4 Utviklingsprosessen	21
2.4.1 Sprint 1 - Kodegjennomgang og Oppstart:	21
2.4.2 Sprint 2 - Oppstartsmøte og GitHub Setup:	21
2.4.3 Sprint 3 - Lokale Oppsett og Prosjektside:	23
2.4.4 Sprint 4 - Oppdatering av Dependencies og PDF API:	23
2.4.5 Sprint 5 - Skinning av Keycloak, Mathesar Integrasjon, Videreutvikling av PDF-API og Betalingssystem:.....	24
2.4.6 Sprint 6 - Administrasjonsgrensesnitt, Database Backup og API-endepunkter: ...	25
2.4.7 Sprint 7 - Videre utvikling av betalingssystem og medlemskapssystemet:.....	26
2.4.8 Sprint 8 - Klargjøre medlemskapssystemet, Systemfeilsøking og Videreutvikling av Betalingssystemet og PDF-modulen:	27
2.4.9 Sprint 9 - Utvikling av e-postsystem og sammenkobling av betalingssystemet ..	29
2.4.10 Sprint 10 – Klargjøring for produksjon:	31

2.5 Kravspesifikasjonen og dens rolle	32
2.5.1 Endringer i kravspesifikasjonen	32
2.5.2 Kravspesifikasjonens rolle i design og implementering.....	33
2.5.3 Samsvar mellom krav og løsning.....	33
3. Produktdokumentasjon	35
3.1 Innledning	36
3.2 Systemarkitektur.....	36
3.2.1 Teknologistabel (Tech Stack)	37
3.3 Kravspesifikasjon	38
3.3.1 Funksjonelle krav	38
3.3.2 Ikke-funksjonelle krav	41
3.4 Medlemskapssystemet	42
3.4.1 Forord.....	42
3.4.2 Oversikt over komponenter medlemskapssystem	43
3.4.3 Brukerflyt.....	44
3.4.4 Frontend.....	46
3.4.5 Backend	48
3.4.6 Brukergrensesnitt (UI).....	55
3.4.6.1 EFN startside (medlemsportal).....	55
3.4.6.2 Digital medlemskort.....	55
3.4.6.3 Brukerinnlogging.....	56
3.4.6.4 Brukerregistering.....	58
3.4.6.5 Verifisering av bruker	60
3.4.6.6 Glemt passord	61
3.4.6.7 Administratorinnlogging.....	62
3.4.6.8 Administrative grensesnitt (Keycloak)	63
3.4.6.9 Mathesar grensesnitt for innlogging	64
3.4.6.10 Mathesar administrasjon av medlemsdata	65
3.4.6.11 Tabelloversikt i Mathesar	68
3.5 Betalingssystemets oppdeling	69
3.5.1 Mappestruktur og bruk av Docker	69
3.5.2 Frontend.....	70

3.5.3 Backend	71
3.5.4 PDF-modul	74
3.5.5 Postgresql db	75
3.5.6 Sikkerhet i backend.....	76
	79
4. Testdokumentasjon	79
4.1 Introduksjon.....	80
4.2 Hvorfor testing er viktig for dette prosjektet.....	80
4.3 Tester.....	80
4.3.1 Tester I Medlemskapssystemet	80
4.3.1.1 Integrasjontest av backup og gjenoppretting av live-database	80
4.3.1.2 Integrasjontest av e-postutsending ved nytt betalt medlem	81
4.3.2 Tester I Betalingssystemet.....	82
4.3.2.1 Enhetstester betalingssystemet.....	82
4.3.2.2 Stresstest av PDF-APIet.....	84
4.4 Testevaluering.....	85
	86
5. Avsluttende del.....	86
5.1 Diskusjon om faglige utfordringer og valg	87
5.1.1 Valg I Medlemskapssystemet	87
5.1.2 Valg I Betalingssystemet.....	88
5.2 Refleksjon	91
5.3 Læringsutbytte	92
5.4 Videreutvikling	93
5.4.1 Betalingssystemet	93
5.5 Konklusjon	94
6. Referanseliste	96

1. Innledning

I dette kapittelet gir vi en oversikt over bachelorprosjektet vårt. Vi starter med en presentasjon av prosjektgruppen, oppdragsgiver og skoleveileder. Videre beskriver vi oppdraget, bakgrunnen for prosjektet og den nåværende situasjonen. Til slutt går vi gjennom målene for prosjektet, forventede gevinstene og eventuelle begrensninger som påvirker arbeidet.

1.1 Presentasjon av gruppen

Vi er en gruppe på fire studenter som har samarbeidet gjennom bachelorprosjektet. To av oss har jobbet sammen på alle gruppeoppgaver gjennom studiet, noe som har gitt oss god erfaring med samarbeid og oppgavedeling. I tillegg kjente tre i gruppen hverandre fra tidlig i studieperioden, noe som har bidratt til et godt arbeidsmiljø og effektiv kommunikasjon. Dette tette samarbeidet har vært en styrke under arbeidet med bacheloroppgaven, spesielt når det gjelder å fordele oppgaver og utnytte hverandres sterke sider.



Adrian Hoff har en sterk interesse for webutvikling, både innen frontend og backend. I dette prosjektet var hovedfokuset på medlemskapssystemet, som inkluderte frontend-komponenter, men der en betydelig del av arbeidet var rettet mot backend-funksjonalitet.

Adrian Hoff

Informasjonsteknologi
374946@oslomet.no



Ask liker problemløsning og har hovedinteresse for backendutvikling og å finne kreative løsninger på problemer. Han har primært jobbet på backend og planlegging på betalingsløsningen.

Ask Norli

Informasjonsteknologi
s374959@oslomet.no



Simen Thams har bakgrunn som Siviløkonom og fra konsulentbransjen. Han har vært veldig interessert i Full-Stack utvikling og har et brennende engasjement for å lære seg å lage gode ERP- og Fintech løsninger for næringslivet. I dette prosjektet har han fokusert mest på betalingssystemet, blant annet utforming av PDF-modulen og sørge for at det følger etablerte forretningsstandarder.

Simen Thams

Informasjonsteknologi
s374935@oslomet.no



Adina sin hovedinteresse er webutvikling. Hun har primært jobbet med utviklingen av medlemskapssystemet, med hovedfokus på backend. I tillegg har hun bidratt til utviklingen av frontend for betalingssystemet

Adina Heia

Informasjonsteknologi
s374989@oslomet.no

1.2 Presentasjon av oppdragsgiver

Organisasjonen vi har bachelorprosjekt for er Elektronisk Forpost Norge (EFN). EFN er en Ideell organisasjon (NGO) som jobber for en rekke digitale rettigheter, deriblant ytringsfrihet, åpne standarder(kildekode), frihet fra overvåkning, åpenhet- og tilgjengelighet av offentlig finansiert forskning og data, samt bruker-kontrollert software og misbruk av copyright-lover ("Member Spotlight: EFN", u.å.).

EFN jobber i hovedsak politisk og påvirker politikken rundt menneskers rettigheter på nett både i Norge og i andre land. De har delt at de har engasjert seg i flere av EU's nyere forordninger innenfor personvern og kunstig intelligens, slik som den nye KI-forordningen. De har også involvert seg i DNS-domenebeslaget rundt popcorntime.no.

I EFN har vi jobbet tett opp imot Tom Fredrik Blenning og Bjørn Remseth. Ved oppstart var Tom Fredrik daglig leder, men han var på vei til å fratre ledertillingen. Han har dog vært vår hovedkontakt opp-i-mot EFN. Bjørn Remseth har laget det initielle medlemskapssystemet til EFN, og har vært en naturlig sparringspartner for teamet som har jobbet med medlemskapssystemet.

1.3 Oppdrag

1.3.1 Bakgrunn for oppdraget - dagens situasjon

Elektronisk Forpost Norge (EFN) benytter i dag et medlemskapssystem som fungerer som en MVP (Minimum Viable Product). Systemet gir grunnleggende funksjonalitet for medlemsregistrering, betaling og oppfølging. Medlemmer kan registrere seg gjennom plattformen membershipdb.efn.no og betale medlemskontingensten via Vipps, som er hoved betalingsløsningen i systemet.

For medlemmer som ikke ønsker å bruke Vipps, finnes det en alternativ løsning der betaling kan gjøres manuelt via bankoverføring. Brukere må selv oppgi sin e-postadresse som referanse i betalingsdetaljene for at innbetalingen skal registreres korrekt. For internasjonale medlemmer er det også mulig å betale gjennom en IBAN-overføring til EFNs bankkonto. Denne prosessen er manuell og krever ekstra oppfølging for å sikre riktig registrering av betalinger, spesielt når betalingen gjøres på vegne av andre.

Mens systemet har vært tilstrekkelig for å starte medlemsregistrering og kontingentinnkreving, har det flere begrensninger:

- **Begrenset fleksibilitet:** Brukerne er avhengige av enten Vipps eller manuell overføring.
- **Manuell oppfølging:** Betalinger som gjøres utenfor Vipps må behandles manuelt, noe som fører til økt risiko for feilregistreringer og høy arbeidsbelastning.
- **Ingen automatisering:** Fakturering og betalingsoppfølging mangler automatiserte løsninger, noe som gjør det tidkrevende for administrasjonen å håndtere medlemstransaksjoner.
- **Brukeropplevelse:** Betalingsprosessen kan oppleves som tungvint, særlig for brukere som velger alternativene utenfor Vipps.

Disse utfordringene betyr at dagens løsning ikke dekker EFNs behov for et moderne og brukervennlig medlemskapssystem som kan vokse med organisasjonen.

1.3.2 Oppdraget

Da vi kom i kontakt med EFN for høring av et potensielt bachelorprosjekt, ble behovet for et nytt medlemskapssystem raskt hovedforslaget for bacheloroppgaven. EFN hadde allerede påbegynt utviklingen av et nytt medlemskapssystem da vi startet på prosjektet.

Under vårt første møte presenterte oppdragsgiver en overordnet modell (vist i figur 3, s.27) som illustrerte hvordan han så for seg at de ulike komponentene i medlemskapssystemet skulle samhandle - spesielt med tanke på medlemsregistrering og betalingsflyt. Denne modellen ble et viktig utgangspunkt for arbeidet vårt, og fungerte som en rettesnor gjennom prosjektperioden.

Oppdraget besto av to hoveddeler. Den første var å videreutvikle og ferdigstille medlemskapssystemet, som på tidspunktet vi overtok det, inneholdt grunnleggende funksjonalitet, men manglet flere sentrale komponenter før det kunne tas i bruk. Den andre delen var å lage et helt nytt betalingssystem som kunne brukes sammen med systemet. Dette betalingssystemet skulle støtte både vanlige fakturaer og kontantfakturaer, og den skulle utvikles som åpen kildekode.

Oppsummert gikk oppdraget ut på å fullføre et allerede påbegynt medlemskapssystem og utvikle en tilhørende betalingsløsning med fleksibel betalingsstøtte, basert på prinsippene for åpen kildekode og gjenbrukbarhet.

1.3.3 Rammebettingelser og mål

Rammebettingelser

Utviklingsarbeidet i prosjektet ble påvirket av flere rammebettingelser, både tekniske og organisatoriske. En sentral utfordring i starten var å definere konkrete og langsiktige mål, da vi arbeidet innenfor et stramt tidsperspektiv og med begrenset teknisk kompetanse. Dette førte til at vi måtte jobbe iterativt, spesielt med betalingssystemet, der vi startet med noen få overkommelige mål og la til flere etter hvert som delmål ble fullført.

For betalingssystemet var det et krav at all kode skulle utvikles under en åpen lisens, og at løsningen skulle være enkel å ta i bruk for andre organisasjoner. Dette var viktig for å sikre at systemet kunne videreutvikles og integreres med medlemskapssystemet. For å muliggjøre dette, var det avgjørende at koden ble strukturert og dokumentert på en måte som gjorde det enkelt for andre utviklere å sette seg inn i arbeidet.

For medlemskapssystemet var det viktig å videreutvikle løsningen basert på teknologiene som allerede var tatt i bruk i det eksisterende systemet. Dette innebar å benytte samme verktøy og programmeringsspråk som allerede var i bruk. Blant de sentrale teknologivalgene var Docker for containisering, PostgreSQL som databasesystem, Keycloak for autentisering, samt React som rammeverk for frontend-utvikling.

En teknisk forutsetning for medlemskapssystemet var at det krevde mye RAM for å kjøre effektivt i Docker. Derfor måtte alle gruppemedlemmer ha nok tilgjengelig RAM og maskinvare som støttet lokal kjøring av systemet.

Hovedmål

Hovedmålsettingen med prosjektet var å skape et fleksibelt og brukervennlig system som tilfredsstiller behovene til både EFN og deres medlemmer. Prosjektet bestod av to hoveddeler: å videreutvikle medlemskapssystemet til produksjonsklar tilstand, samt å utvikle et tilhørende betalingssystem som kunne håndtere ulike betalingsformer.

Delmål: Medlemskapssystem

- Oppdatere avhengigheter
- Tilpasse utseendet og designet i Keycloak
- Implementere et verktøy for enkel oversikt over medlemsdatabasen
- Verifisere at backup-løsning fungerer og dokumentere prosessen
- Klargjøre frontend for produksjon uten debug-modus
- Lage en metode eller skript for å sende kvittering på e-post når et medlem betaler for medlemskap

Delmål: Betalingssystem

- Sette opp og finne ut av struktur (backend)
- Utvikle god metode for verifisering av faktura og betalingsinformasjon
- Lage en PDF-generator som behandler data og returnerer faktura basert på input
- Lage en veldig enkel frontendside hvor vi kan manuelt endre på betalingsstatus
- Lage funksjonalitet for generering av kontantfaktura i frontendsiden
- Systemet skal være modulært og kunne deployes på Docker
- Sette opp et API-endepunkt som kan brukes for PDF- og fakturagenerering

Læringsmål

For gruppemedlemmene var prosjektet en verdifull mulighet til faglig og praktisk utvikling.

Vi ønsket å oppnå erfaring med utvikling av modulære systemer ved bruk av moderne webteknologier og Docker. I tillegg ga arbeidet innsikt i hva det innebærer å jobbe i et åpent kildekodemiljø med særlig fokus på personvern og dataminimering.

Prosjektet ga også erfaring med gruppearbeid i en realistisk utviklingskontekst, hvor vi måtte samarbeide tett, fordele ansvar og løse tekniske og organisatoriske utfordringer i fellesskap.

1.3.4 Forventede gevinster

Ved ferdigstillelse av prosjektet forventet vi flere gevinster på ulike områder. For EFN som organisasjon ville et nytt system bety redusert manuelt arbeid gjennom automatiserte prosesser for medlemsregistrering, fakturering og betalingsoppfølging. Redusert manuelt arbeid vil redusere risiko for manuelle feil i medlemsregistrering og betalingsprosessen. Samtidig vil systemet tilby administrative verktøy som vil gjøre det lettere for organisasjonen å holde kontroll på medlemmene i databasen.

For medlemmene ville gevinstene omfatte en enklere registreringsprosess med et forbedret brukergrensesnitt. Etter hvert som betalingssystemet blir integrert i medlemskapssystemet, vil det kunne tilbys flere betalingsalternativer slik at man dekker brukernes ulike behov og preferanser. Medlemmene vil også få økt tillit til systemet gjennom automatiske betalingsbekreftelser sendt til de på epost.

Den litt mer overordnede interessegevisten var potensiale for å utvikle en Open Source løsning som flere ideelle organisasjoner i Norge kan benytte seg av. Dette er veldig i EFN sin ånd som forkjemper for åpen kildekode.

2. Prosessdokumentasjon

2.1 Innledning

Denne delen beskriver de ulike utviklingsfasene i prosjektet vårt for EFN og refleksjoner rundt de faglige utfordringene vi har møtt underveis. Vi går gjennom de viktigste valgene vi har tatt angående systemets oppbygging og funksjonalitet, hvilke verktøy vi har brukt, hvilke tekniske problemer som har vært mest krevende, og hvordan samarbeidet med oppdragsgiveren har utviklet seg over tid. Til slutt reflekterer vi over mulige forbedringer og videreutvikling av systemet.

Prosjektet innebærer forbedring av EFNs medlemskapssystem og utvikling av et separat, open-souce betalingssystem som på sikt kan videreutvikles og benyttes av andre organisasjoner. Arbeidet har vært iterativt, med løpende tilpasninger basert på testing og veiledning fra EFN.

2.2 Planlegging og metode

Arbeidsprosessene vi har valgt å bruke er kombinasjon av Scrum og Kanban. En kombinasjon av Scrum og Kanban gir oss en balanse mellom struktur og fleksibilitet som vi mener er viktig for en bacheloroppgave. De faste møtene og iterasjonene fra Scrum gir oss et rammeverk for fremdrift, mens den visuelle oppgavehåndtering og fleksibiliteten fra Kanban lar oss tilpasse oss endringer underveis.

Ozkan et al. (2022) publiserte et litteratursammendrag hvor de gjennomgikk fagfellevurderte studier på effektiviteten av Scrum og Kanban (og en blanding av de to). Ifølge forfatterne så beskrives gjerne Scrum som litt mer definert og tydeliggjort. Med veldefinerte «sprinter» eller bolker av jobber. Det passer bedre for større team og for å jobbe mot en deadline. Vi har dog i enighet med EFN kommet frem til at å definere tidsaspekter når vi alle er ferske utviklere blir utfordrende. Vi har underveis vært nødt til å gjøre tilpasninger og justeringer. Vi snakker ofte om såkalte «must have»- og «nice-to-have»'s. Kanban er i følge Ozkan et al. (2022) mer fleksibel og tilpasningsdyktig over Scrum. Det er også litt mer fokus på arbeid og leveranse. Vi ser styrkene med begge verktøyene og arbeidsmetodene, men vi har funnet ut at en blandet tilnærming er det som har fungert best for oss. Forfatterne beskriver at Kanban ser ut til å ha et knapt bedre resultat enn Scrum, og at flere selskaper tilnærmer seg mer Kanban-metoden, men nevner at en kombinasjon av disse er å anbefale. Vi har bestemt oss for å gå for denne anbefalte tilnærmingen.

Vi har ukentlige «Scrum-inspirerte»-møter hver mandag hvor vi hver for oss gjennomgår tre hovedspørsmål:

- Hva har du gjort siden sist?
- Hva planlegger du å gjøre i løpet av uken?
- Er det noe som hindrer deg?

Tilbakemeldinger fra oppdragsgiver har vært avgjørende for å løse ukens utfordringer, og planlegge og sette neste sprint. Gjennom møter og skriftlige tilbakemeldinger har vi fått innsikt i EFNs spesifikke behov, noe som har hjulpet oss med å prioritere oppgaver og gjøre nødvendige justeringer underveis. Disse ukentlige møtene har vært essensielt for å holde oss på sporet.

Planleggingen startet med en kartlegging av eksisterende systemer og oppdeling av prosjektet i mindre oppgaver. Vi brukte GitHub Issues til å definere og følge opp oppgaver, som fungerer som en Kanban-tavle. Her organiserer vi oppgavene i ulike faser, som "To do", "In progress" og "Done". Denne tilnærmingen gir oss en visuell oversikt over arbeidet og gjør det enklere å følge med på fremdriften.

Kommunikasjonen internt i gruppen har primært foregått via Discord, mens vi har brukt Signal til å holde kontakt med oppdragsgiver. Ukentlige statusmøter med EFN har gitt oss mulighet til å diskutere utfordringer, avklare spørsmål og justere kurset etter behov.

Vi jobber ofte med flere oppgaver parallelt, avhengig av oppgavenes størrelse og kompleksitet. Små oppgaver håndteres samtidig, mens større oppgaver får mer fokus. Vi har høy grad av fleksibilitet når det gjelder endringer. Krav og oppgaver kan justeres underveis, og vi avveier prioriteringer løpende.

I løpet av prosjektet har vi også måttet sette oss inn i nye teknologier. Blant annet har vi måttet lære mer om Docker for å forstå hvordan systemet kjører i containere, samt håndtere Keycloak for autentisering og Mathesar for databaseadministrasjon.

2.3 Teknologi og verktøy vi har brukt

Dagbok

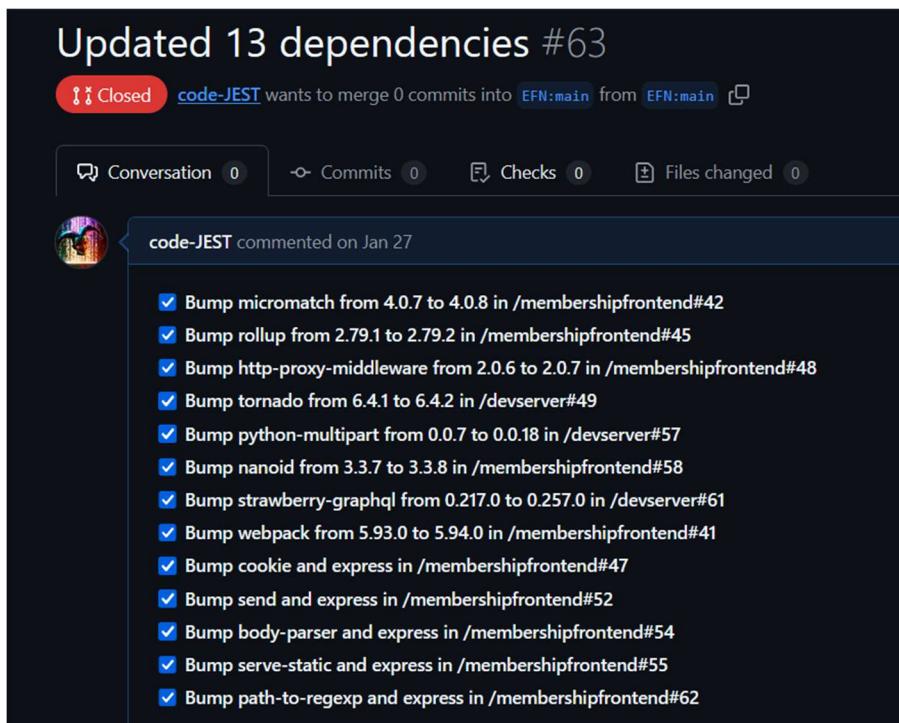
Vi har brukt en digital dagbok til å loggføre hva som gjøres, utfordringer og annen relevant informasjon. Denne var delt med hele gruppen gjennom OneDrive, og alle hadde som

oppgave å passe på å holde den oppdatert underveis i arbeidet. Terskelen for hva som kunne skrives her var veldig lav så det var enkelt å bruke noen minutter hver dag til å dokumentere fremdrift og hendelser.

Dependabot

Dependabot er et verktøy integrert i GitHub som automatisk holder prosjektets avhengigheter oppdatert og varsler om kjente sikkerhetssårbarheter. Det fungerer ved å overvåke avhengighetsfilene i prosjektet og foreslå oppdateringer når det finnes nye, sikrere eller mer stabile versjoner (Github, 2025).

I starten av prosjektet brukte vi Dependabot aktivt for å oppdatere alle avhengigheter hvor det fantes nye versjoner (se figur 1). Flere avhengigheter var utdatert og krevde oppgradering, etterfulgt av grundig testing for å verifisere at de fungerte korrekt. Den varslet oss automatisk dersom det ble oppdaget nye sikkerhetsproblemer knyttet til noen av avhengighetene våre, slik at vi raskt kunne reagere og redusere risiko for sårbarheter i koden vår.



Figur 1 Lukket pull request som viser oppdaterte avhengigheter

GitHub

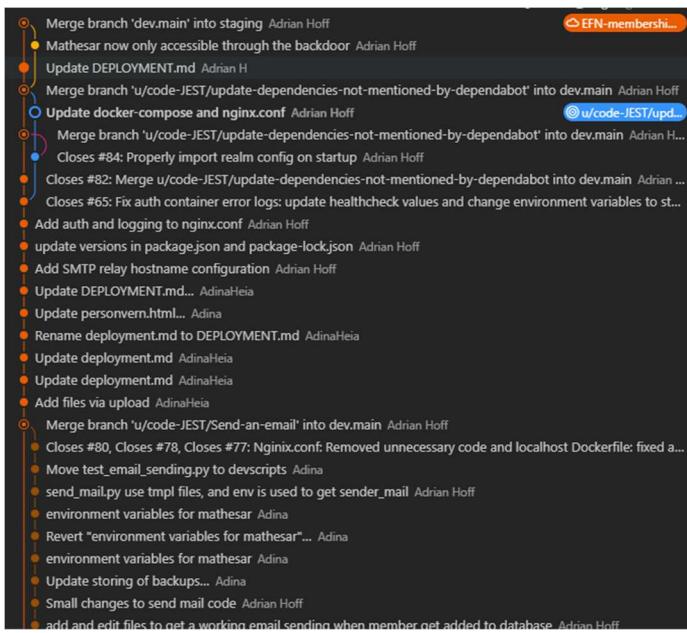
GitHub er en plattform for versjonskontroll og samarbeid, bygget rundt Git. Den gjør det mulig for utviklere å lagre, spore og samarbeide om kodeprosjekter i sanntid. GitHub gir også

funktionalitet for oppgavehåndtering, dokumentasjon og automatisering, og er mye brukt i både utdanning og næringsliv.

Vi har brukt private repositories til både medlemskapssystemet og betalingssystemet, og organisert oppgaver gjennom Issues og Milestones. Dette har gjort det lettere å samarbeide og holde oversikt over endringer i kodebasen.

Begge repositoriene fulgte en GitLab Flow grenestrategi, hvor main-grenen skulle bli stabil og urørt, spesielt for medlemskapssystemet som var koblet til produksjonsmiljøet. GitLab Flow gir god struktur i utviklingen, gjør det lett å skille mellom hva som utvikles, testes og er klart for produksjon. Det gjør det trygt å jobbe parallelt og sørger for at kun testet kode havner i produksjon. GitLab Flow kan beskrives som en hybrid mellom Git Flow og GitHub Flow, hvor utviklingsarbeidet støttes av separate grener for produksjon, staging og utvikling. Funksjoner utvikles i egne feature branches som deretter flettes inn i utviklingsgrenen, før de flyttes videre til staging for testing og til slutt til produksjon (Gowda, 2022).

Utviklingen foregikk i egne feature-brancher knyttet til en eller flere issues. Når en oppgave var ferdigstilt, ble endringene først integrert i en felles dev.main-gren (se figur 2). Etter at funksjonaliteten var verifisert der, ble den videreført til en staging-gren. Staging-grenen benytter et eget staging-miljø på serveren for å kunne teste om alt funker som det skal på en kjørende server. Kun når all funksjonalitet var grundig testet og godkjent i staging, ble endringene merget til main-grenen for produksjon.



Figur 2 Skjermbilde av vår Github branch

Keycloak

Keycloak er en åpen kildekode-løsning for identitet og tilgangsstyring (Keycloak, u.å.). Det brukes for å håndtere autentisering, autorisering, og sikker håndtering av brukere og roller. Keycloak støtter moderne autentiseringsprotokoller som OAuth2, OpenID Connect og SAML, og tilbyr funksjoner som Single Sign-On, tofaktorautentisering, og mulighet for integrasjon med eksterne identitetsleverandører som Google og Microsoft.

Keycloak har vært sentralt i autentiseringsløsningen til medlemskapssystemet. Det var allerede en del av EFNs system før vi startet prosjektet, men vi har jobbet med å tilpasse utseendet og funksjonaliteten gjennom skinning og integrasjon.

PostgreSQL

PostgreSQL er et objektrelasjonelt databasesystem med fokus på robusthet og fleksibilitet. Den brukes både i medlemskapssystemet og betalingssystemet for lagring av brukerdata og betalingsinformasjon (PostgreSQL, 2025)

React

React har blitt brukt til utvikling av frontend for medlemskapssystemet. Dette rammeverket ble valgt fordi det er moderne, effektivt og gir god modularitet (React, 2025)

Docker

Docker har vært avgjørende for containerisering av applikasjonene våre. Vi brukte Docker til å sikre at utviklingsmiljøet vårt var likt på tvers av ulike systemer og til enklere distribusjon av tjenestene vi utviklet (Docker, 2025)

Python

Vi valgte Python til backend-utviklingen fordi vi hadde god kjennskap til språket fra tidligere prosjekter, og det tilbyr et bredt utvalg av nyttige biblioteker (Python Software Foundation, 2025)

Visual Studio Code

VS Code har vært vårt hovedutviklingsmiljø gjennom hele prosjektet. Med innebygd støtte for Git-integrasjon, terminal, og et bredt utvalg av utvidelser for blant annet koding, feilsøking og språkstøtte, har det bidratt til en mer effektiv og oversiktlig utviklingsprosess (Microsoft, 2025)

Kommunikasjonsverktøy

Discord har fungert som vårt primære verktøy for intern kommunikasjon. Slik som daglige diskusjoner, deling av notater og samarbeid i sanntid. Signal ble brukt til noe av det tilsvarende, men mot oppdragsgiveren. Signal ble mye brukt pga sitt fokus og renommé for datasikkerhet og anonymitet (Signal Messenger, u.å.). For videosamtaler brukte vi Redpill Linpro sin videokonferanse-tjeneste som er åpen kildekodet, og var det foretrukne digitale «møterommet» til oppdragsgiver.

KI-verktøy

Under utviklingsfasen har vi benyttet oss av KI-verktøy slik som ChatGPT, DeepSeek, og Grok da det kom ut. De to sistnevnte ble minimalt brukt når det kom ut, mest for å teste ut og sammenligne, men i all hovedsak har vi benyttet oss av ChatGPT som assistent, opplæringsverktøy, sparringspartner, og for å fikse bugs og produsere skallkode. Vi har hatt dialog med oppdragsgiver om fornuftig bruk av KI-verktøy og har hatt ganske frie tøyler ettersom det er et åpent kildekode-prosjekt, og vi ikke har introdusert noen sensitive data. Den overordnede filosofien har vært å ikke bruke noe vi ikke forstår. Dette har vært overveiende vurdert gjennom utviklingsfasen.

Mathesar

Mathesar ble valgt for sikker håndtering av databaser i medlemskapssystemet. Det gir en brukervennlig måte å administrere PostgreSQL-databasen uten behov for direkte SQL-forespørsler. En særlig fordel med Mathesar er støtten for manuell registrering av betalinger fra andre kilder enn Vipps, noe som gir fleksibilitet i betalingsløsninger (Mathesar, u.å.)

WSL 2 med Ubuntu

WSL er en funksjon i Windows som gjør det mulig å kjøre et Linux-operativsystem direkte i Windows, uten behov for en virtuell maskin eller dual-boot. Den nyeste versjonen WSL 2, inkluderer en ekte Linux-kjerne og gir full støtte for Linux-filsystemet, samt bedre ytelse og kompatibilitet sammenlignet med den første versjonen (Docker Documentation, 2025)

For de av oss som ikke bruker Mac, har vi brukt WSL for å få tilgang til Linux-terminaler og kjøre utviklingsverktøyene våre på en mer effektiv måte.

SSH-tunneling

For å få tilgang til kritiske tjenester på serveren måtte vi etablere en SSH-basert SOCKS5-proxy. Dette var spesielt viktig for å få tilgang til Keycloak, som var plassert bak en «backdoor» på serveren og derfor ikke var direkte tilgjengelig fra internett.

Denne løsningen var satt opp for å sikre Keycloak mot uautorisert tilgang, ettersom en kompromittert Keycloak-instans kunne utgjøre en alvorlig fare for hele systemet. Selv om Keycloak krypterer passord med saltet hashing i sin database, ville direkte tilgang til administrasjonsgrensesnittet kunne gi angripere mulighet til å endre autentiseringslogikk, skape nye administrative brukere eller manipulere sikkerhetspolicyer.

EFN Skyløsning

Vi brukte EFN Skyløsning for å samle alle prosjektdokumenter på ett sted. Her lagret vi grafer, teknisk dokumentasjon, tegninger og forklaringer som var viktige for prosjektet. Det gjorde at alle i gruppen hadde tilgang til oppdatert informasjon når som helst, uansett hvor vi var. Dette var spesielt nyttig når vi jobbet hjemmefra eller på ulike steder, siden alle alltid kunne se de nyeste versjonene av dokumentene våre.

Draw.io og sequencediagram.org

Vi har benyttet draw.io og sequencediagram.org for å lage diagrammer (draw.io, u.å., SequenceDiagram.org, u.å.). Diagrammene har vært et viktig hjelpemiddel for å forstå

hvordan de ulike modulene henger sammen og for å få et helhetlig overblikk over oppgaven vi står ovenfor. De har også gjort det enklere å forklare ideer til hverandre gjennom visualisering, og finne ut av hvilke funksjoner og moduler som er nødvendige i løsningen og hvordan disse skal samhandle.

2.3.1 Oppsett av prosessverktøy

For å organisere oppgaver og følge fremdriften brukte vi GitHub Issues. Dette gjorde det enklere å dele oppgaver, spore hva som var fullført, og holde oversikt over kommende arbeid. Hver oppgave ble dokumentert som en issue. Vi benyttet to separate GitHub-repoer: betalingssystemet, som fokuserte på betalingsfunksjonalitet, og medlemskapssystem, som dekket frontend, adminløsninger for databasen, database-testing osv. Denne oppdelingen gjorde det lettere å holde systemene adskilt og arbeidsflyten ryddig.

2.4 Utviklingsprosessen

Prosjektet vårt hos EFN har gjennomgått flere utviklingsfaser, hvor vi gradvis har fått en dypere forståelse av medlemskapssystemet og de tilhørende komponentene.

2.4.1 Sprint 1 - Kodegjennomgang og Oppstart:

I starten av prosjektet gjennomførte vi en grundig kodegjennomgang for å identifisere sensitiv data og forstå strukturen i kodebasen. Dette innebar å analysere eksisterende funksjonalitet, dokumentasjon og avhengigheter for å få en helhetlig oversikt. Vi kartlegget miljøvariabler og noterte oss eventuelle svakheter som kunne påvirke videre utvikling.

Vi ble informert om at Docker-systemet krevde minimum 16 GB RAM (helst 32 GB) for stabil kjøring av medlemskapssystemet. Dette skapte utfordringer i begynnelsen for de som hadde mindre minne ettersom systemet enten ikke startet eller kjørte veldig tregt. Løsningen ble å bytte til en PC med mer minne eller oppgradere maskinvaren.

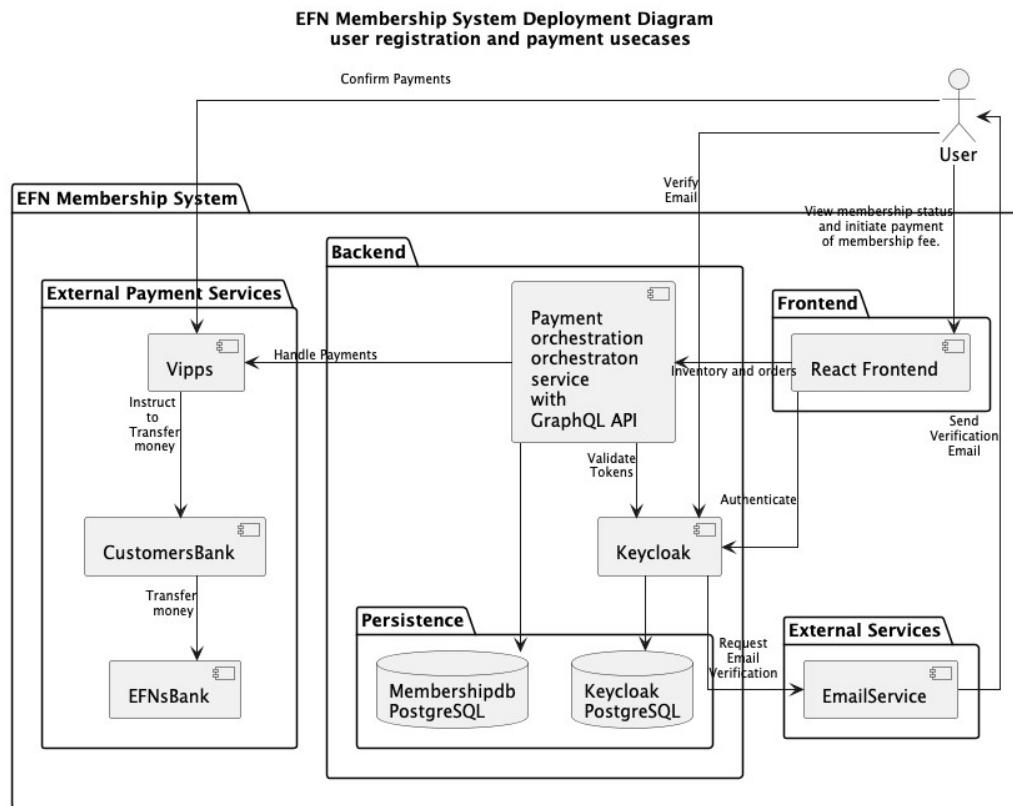
2.4.2 Sprint 2 - Oppstartsmøte og GitHub Setup:

I uke 2 gjennomførte vi et formelt oppstartsmøte hjemme hos daglig leder i EFN. Under dette møtet satte vi opp GitHub-brukere med to-faktor-autentisering (2FA), SSH-nøkler, og tilgang

til medlemskapssystemets repository. Vi gjennomgikk også kodebasen for å danne oss et helhetlig bilde av prosjektet og identifisere mulige utfordringer.

En av de største utfordringene vi møtte var mengden informasjon og kompleksiteten i systemet. Til tross for en detaljert forklaring, slet vi med å se sammenhengen mellom de mange filene, skriptene og mappene. Dette ble forsterket av at ingen av oss hadde noe særlig erfaring med Docker, noe som gjorde det vanskelig å forstå systemets struktur og hvordan de ulike delene hang sammen.

For å løse denne utfordringen hadde EFN laget et diagram av systemet for å lettere gi oss forståelse av det. Dette diagrammet viser EFN Membership System (se figur 3), med fokus på brukerregistrering og betalingsprosesser, og det hjalp oss med å begynne å se hvordan systemet faktisk fungerer. Dette visuelle overblikket ga oss en større forståelse av systemets arkitektur sammenlignet med å bare se på mange filer og mapper i kodebasen. Vi kunne nå se hvordan de forskjellige komponentene henger sammen og kommuniserer.



Figur 3 Diagram av EFN Membership System – utbedret av Bjørn Remseth i EFN.

2.4.3 Sprint 3 - Lokale Oppsett og Prosjektside:

Vi ble vi delt inn i to grupper med separate oppgaver, den ene gruppen jobbet med å få medlemskapssystemet opp å kjøre på sine lokale maskiner, mens den andre jobbet med å lage en prosjektside for gruppen (<HTTPs://webinnviklerne.efn.no/index.html>). For å håndtere utfordringene med det komplekse systemet brukte vi parprogrammering, der to og to samarbeidet om samme kodebase.

Gruppen som arbeidet med medlemskapssystemet, møtte flere tekniske utfordringer underveis. En av de største hindringene var å håndtere forskjeller i operativsystemer, spesielt siden oppdragsgiver primært brukte Mac mens vi hovedsakelig brukte Windows. Valget falt på Windows Subsystem for Linux 2 (WSL 2) med Ubuntu, som lot oss kjøre Linux-kommandoer og verktøy direkte fra Windows. WSL 2 forenkler kjøring av Linux-distribusjoner på Windows ved å forbedre ytelse og filsystemtilgang (Docker Documentation, 2025). WSL 2 eliminerte problemer med filsystem-inkompatibilitet og scriptutførelse, noe som reduserte tiden brukt på feilsøking av plattformspesifikke problemer. Som en tilleggsgevinst fikk teammedlemmene verdifull erfaring med Linux-kommandoer.

2.4.4 Sprint 4 - Oppdatering av Dependencies og PDF API:

En gruppe fortsatte med å oppdatere dependencies i medlemskapssystemet, mens den andre begynte utviklingen av et faktura-PDF API. Under arbeidet med Docker-prosjektet fikk vi verdifull hands-on erfaring med containere. I begynnelsen brukte vi --build flagget ved hver oppdatering for å sikre at endringene ble implementert. Dette var betydelig ressurskrevende ettersom det tvinger systemet til å bygge bildene på nytt fra bunnen, kompile all kode og installere alle avhengigheter på nytt. Etter hvert som vi ble mer fortrolige med Docker-miljøet, oppdaget vi en bedre løsning. Mange endringer kunne implementeres effektivt ved å bare bruke “Docker compose up” og “Docker compose down” kommandoene. Denne metoden gjenbruker eksisterende bilder og tar vesentlig mindre CPU, minne og tid. Erfaringen ga oss en dypere forståelse av hvordan Docker håndterer applikasjoner og deres avhengigheter.

I oppstartsfasen begynte vi med å lage et enkelt fakturaprogram som tok inn noen parametere og returnerte en ferdig PDF. Etter innspill fra oppdragsgiveren vår fikk vi anbefalt å bruke UBL (Universal Business Language) som format for å beskrive fakturaen. Han foreslo også

at vi burde tolke UBL-strukturen med Pydantic, og bruke FastAPI med Uvicorn for å bygge API-et. Med det som utgangspunkt satt vi sammen en grunnleggende løsning der brukeren kunne sende inn en UBL-XML via en POST-forespørsel i FastAPI, og få tilbake en enkel, automatisk generert PDF. Det var et minimalistisk oppsett, men det ga oss et godt teknisk rammeverk å bygge videre på.

2.4.5 Sprint 5 - Skinning av Keycloak, Mathesar Integrasjon, Videreutvikling av PDF-API og Betalingssystem:

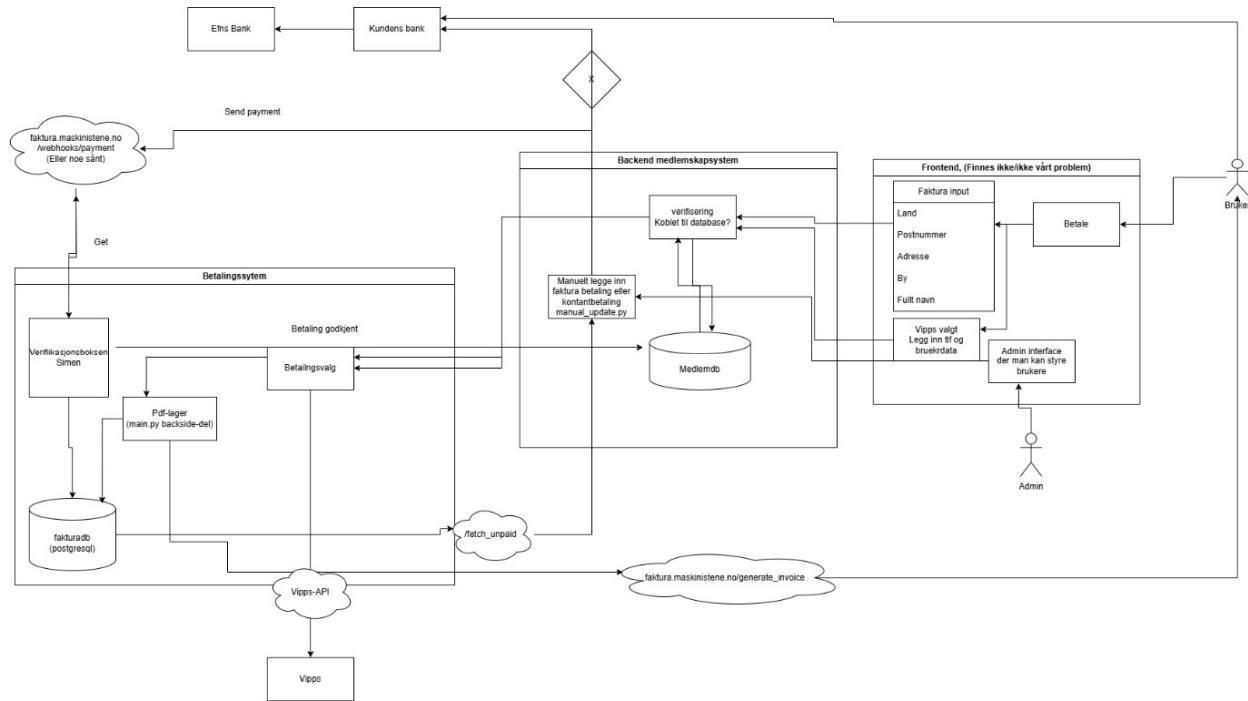
Den ene gruppen begynte å jobbe med integrasjonen av Keycloak-temaet og Mathesar i medlemskapssystemet. Keycloak gir mange muligheter for å tilpasse utseendet gjennom temaer, men det krevde at vi satt oss godt inn i hvordan systemet faktisk er bygd opp. Vi brukte en del tid på å skjønne hvordan tema-filene påvirker brukergrensesnittet og hvordan vi kunne få Keycloak til å automatisk laste inn vårt eget tema ved å mounte det som et volum.

For Mathesar-integrasjonen oppnådde vi raskere fremgang. Så snart vi fikk satt det opp og i gang, begynte vi å koble det til de eksisterende databasene. I tillegg opprettet vi en egen database for funksjonaliteten som hører til Mathesar. Dette ga oss god praktisk erfaring med å sette opp databaseintegrasjon.

PDF-programmet måtte tilpasses slik at det kunne kjøres med Docker-compose. Dette gjorde det mye enklere å starte opp og stoppe systemet raskt. Videre forbedret vi utseende på PDF-en og justerte input parametere for fakturaen. Vi startet også arbeidet med betalingssystemet - et system for å holde styr på betalinger og fakturaer og sikre at folk betaler det de skal. Første steget var å planlegge databasen og definere hvilke verdier og tabeller som var nødvendige.

Vi endte opp med én tabell for betalinger og én for fakturaer. Vi fortsatte med å lage en detaljert plan for hvordan betalingssystemet skulle bygges opp og hvordan det skulle kommunisere med de øvrige delene av medlemskapssystemet. For å planlegge lagde vi et noe forenklet diagram for hvordan betalingssystemet skulle samarbeide med

medlemskapssystemet (se figur 4).



Figur 4 Et førsteutkast diagram av betalingsløsninger

2.4.6 Sprint 6 - Administrasjonsgrensesnitt, Database Backup og API-endepunkter:

En gruppe begynte med å eksponere Mathesar gjennom en reverse proxy og fokusere på database backup-funksjonalitet. For å være sikre på at den opprinnelige backup-løsningen faktisk fungerte som den skulle, testet vi hvordan den hentet og gjenopprettet databasen. Vi tok ut backup-filer fra staging, og gjenopprettet disse både lokalt og på dev-miljøet. Vi oppdaget at backupene ble lagret direkte i containeren, noe vi justerte slik at de i stedet lagres i en egen mappe utenfor containeren. Dette gjør det enklere å hente ut backupene ved behov, noe oppdragsgiver også ønsket.

Å eksponere Mathesar via en reverse proxy viste seg å være mer utfordrende enn forventet, men ga oss viktig innsikt i hvordan systemene henger sammen. Vi måtte gjøre flere endringer, blant annet i miljøvariabler og Docker-compose-filen, for å få Mathesar til å fungere på sitt eget subdomene.

Parallelt med dette arbeidet ble det satt opp API-endepunkter som henter betalingsinformasjon, som blir prosessert og verifisert av en egen verifiseringsmodul. Disse

endepunktene sender og pusher dataene inn til en database som holder oversikt over betalingsstatusen til fakturaene. For å teste hele use-caset for fakturahåndtering, ble det satt opp en lokal PostgreSQL-database.

Her støtte vi imidlertid på en del utfordringer - spesielt med å få databasen til å kommunisere med Docker-containerne våre. Problemet var at hele systemet vårt var definert i en Docker-compose.yml, mens databasen kun var satt opp lokalt på en maskin utenfor Docker-miljøet.

Et av de mer overordnede problemene vi også støtte på i denne perioden, var hvordan vi jobbet sammen som gruppe. Vi jobbet både individuelt og i par, men ofte på de samme oppgavene. Det førte til at vi brukte mye tid på å sette oss inn i hverandres fremgang og forstå hva som var gjort og ikke gjort. Det ble delt mye informasjon på discord og under møter, men det var fortsatt en utfordring å jobbe effektivt sammen og ikke trække hverandre på tærne.

2.4.7 Sprint 7 - Videre utvikling av betalingssystem og medlemskapssystemet:

Her avsluttet vi parprogrammeringen, nå hadde alle i gruppen fått bedre forståelse av systemet og derfor var det mer effektivt å fordele oss i fire.

Medlemskapssystemet:

En viktig oppgave var å rebase feature-branchen mot main på grunn av mange nye commits. I tillegg ble medlemskapssystemet satt opp på serveren til EFN, men dette forårsaket problemer med skinningen av Keycloak, noe som krevde ytterligere feilsøking. I denne sammenhengen ble også Keycloak oppdatert fra versjon 23.0.3 til 26.1.0. Det ble også jobbet med å bygge frontend-komponenten uten debug-alternativ for å optimalisere ytelsen. Det ble også laget en personvernerklæring som skulle stemme med dataene som ble behandlet i systemet.

Under testing ble det oppdaget problemer med innloggingen, trolig på grunn av react nå kjørte i produksjonsmodus. Vi oppdaget at «client ID» ikke ble hentet riktig under byggingen av prosjektet, noe som forårsaket en feilmelding.

Betalingssystemet:

I betalingssystemet ble det arbeidet med PDF-generering. Dette inkluderte endringer for å håndtere dynamisk oppdatering av fakturautstederinformasjon og forbedring av formateringen i parse-funksjonen. Det ble også satt opp enhetstester for å teste funksjonene.

Det ble gjennomført en demo av PDF-delen av betalingssystemet. På denne demonstrasjonen ble det identifisert flere forbedringsområder. Videre ble det jobbet med å opprette en frontend-side knyttet til backend i medlemskapssystemet, men på grunn av manglende tid ble den ikke implementert inn i medlemskapssystemet. Backenden skulle hente ubetalte fakturaer fra en Postgres-database og vise dem til en administrator som kunne markere dem som betalt. Det ble også fokusert på å oppdatere eksisterende kode for backenden og utvikle frontend-komponenter i React med tilhørende enhetstester

2.4.8 Sprint 8 - Klargjøre medlemskapssystemet, Systemfeilsøking og Videreutvikling av Betalingssystemet og PDF-modulen:

Medlemskapssystemet:

I denne perioden var fokuset på å klargjøre medlemskapssystemet for produksjonsmiljø, noe som tar lang tid fordi feil oppdages i denne prosessen. Vi startet med å implementere personvernerklæringen, som ble utarbeidet forrige uke, i brukergrensesnittet (Keycloak). Dette viste seg å være mer utfordrende enn forventet, da det første forsøket førte til at registeringssiden sluttet å fungere. Etter en del prøving og feiling, kombinert med lesing av Keycloaks dokumentasjon og ressurser på nett, fant vi til slutt en fungerende løsning. Vi endte med å lage en «register.ftl» fil under innloggingsdelen av vårt tilpassede Keycloak-tema. Denne gjenskapte registeringssiden slik den var, men med tillegg av teksten og lenken til personvernerklæringen.

Under testing viste det seg at innloggingen hadde sluttet å fungere igjen. Når man prøvde å logge seg inn ga den en Missing Access Token-feil. Dette førte til at brukeren ble logget inn i Keycloak (dette var synlig i adminpanelet), men ble sendt tilbake til innloggingssiden uten tegn på at innloggingen var vellykket eller at man fikk tilgang til systemet. Vi prøvde å legge til bedre feilhåndtering i App.js for å få mer oversikt over hva som faktisk feilet. Etter mye feilsøking uten resultat prøvde vi å søke hjelp fra oppdragsgiver, men denne hjelpen ga heller ingen resultater. Til slutt fant vi ut at problemet stammet fra Keycloak-opgraderingen vi hadde gjort forrige periode. Vi mistenkte at oppgraderingen kunne ha endret noe i systemet, så vi prøvde å gå tilbake til original versjon 23.0.3 og da fungerte alt som det skulle.

Vi oppgraderte Keycloak gradvis versjon for versjon og tilpasset systemet underveis. For hver oppgradering testet vi nøye at alt fungerte som det skulle. Prosessen var tidkrevende, men helt nødvendig for å unngå at systemet skulle slutte å fungere igjen. Vi kom oss opp til versjon 25.0.4, men valgte å stoppe der, siden neste hopp til 26.0.0 krevde omfattende endringer og versjon 25.0.4 dekket behovene våre.

Etter dette lagde vi en demo-video av medlemskapssystemet, der vi viste hvordan man legger til brukere i databasen, logger inn som medlem for å vise medlemskort, og ser e-postbekrefelse ved konto-oprettelse. Vi viste også administrasjon i Keycloak, inkludert brukerhåndtering og temaendringer, samt hvordan man kan legge til data i Mathesar og overføre data fra et system med debug til et annet. Til slutt viste vi hvordan man tar backup, fjerner og gjenoppretter data.

Betalingssystemet:

Vi jobbet også med å gjøre PDF-modulen reentrant. Det vil si at den skal kunne kjøres uten sideeffekter, som innebefatter blant annet at det skal kunne kjøres på nytt eller samtidig flere ganger uten at det oppstår problemer og/eller uforutsette hendelser og sideeffekter (Reentrancy (Computing), 2025). Dette er viktig fordi vi trenger at modulen kan håndtere flere forespørsler samtidig uten at data blir blandet eller ødelagt, dette er særlig viktig for API-endepunktet. Vi har gjennomført stresstester som noe av det siste vi gjorde i ettertid fra da vi konkluderte kodingen.

I tillegg måtte modulen kunne håndtere API_TOKENS for å tilpasse brukerinnstillinger og generering av fakturaer, slik at fakturaene kunne være spesifikke for hver bruker. Det ble mye problemer med main og PDF-modulen når vi begynte å endre kode i databaseklasser og ekspanderte API-endepunktet for API_TOKENS. Selv å generere faktura basert på null token funket ikke lengre.

I betalingssystemet la vi inn støtte for kontantbetaling, som skal håndteres manuelt via admin-panelet. For å få det til lagde vi en løsning der man kan opprette fakturaer i admin, som kan betales senere når kontantene faktisk kommer inn. Kortfakturaer skal komme inn automatisk utenfra, mens kontantfakturaene blir opprettet av admin når han fysisk mottar pengene.

Vi tok en gjennomgang av koden sammen med oppdragsgiver, og da dukket det opp flere steder i koden der ting kunne ellers burde blitt gjort på en annen måte. For eksempel å bruke

databasens innebygde «autoincrement» istedenfor å finne høyeste id og legge til 1. Kanskje det viktigste vi oppdaget i møte med oppdragsgiver var at programmet ikke var enkelt for en ny bruker å iverksette. Vi snakket om hvordan vi kunne gjøre endringer for å gjøre programmet enkelt å starte for nye brukere.

Vi gikk til verks med å løse problemet og starter med å fjerne programmets avhengighet av miljø filer, som tidligere måtte ligge i hver mappe og inneholdt hver enkelt containers hemmeligheter. Nå holder det med en enkelt Docker-compose.override.yml som inneholder hemmelighetene til alle containerene. For å sikre oppstart ble det lagt til et entrypoint.sh-script som sørget for at containerne startet i riktig rekkefølge ved hver oppkjøring.

2.4.9 Sprint 9 - Utvikling av e-postsystem og sammenkobling av betalingssystemet :

Medlemskapssystemet:

Denne uka jobbet vi med å få på plass funksjonalitet for å sende e-post med kvittering til brukeren etter at de har betalt for medlemskapet sitt. Vi hadde ingen tidligere erfaring med å sette opp et e-postsystem som bruker en SMTP-sattelite host, så vi måtte bruke en del tid på å lese oss opp og forstå hva som faktisk må til. Etter litt prøving og feiling fikk vi sendt en enkel test-epost til oss selv og begynte å undersøke hvor i koden det ville være best å legge inn funksjonen for å sende e-post automatisk.

Etter et møte fikk vi noen gode tilbakemeldinger og endringsforslag til hvordan send_mail.py burde være bygget opp. Vi gjorde den mer fleksibel, slik at den nå bruker .tmpl-filer for all tekst som skal inn i e-posten. Tidligere hardkodet e-postinformasjon ble fjernet, og hele SMTP-konfigurasjonen hentes nå fra miljøvariabler. Dette gjør løsningen både tryggere og mer fleksibel.

Uken etter fikk vi satt opp medlemskapssystemet på staging-serveren. Oppsettet gikk overraskende greit, men da vi begynte å teste hele løsningen i staging-miljøet, dukket det opp en stor bug med innlogging og registrering via frontend. Vi mistenkte at feilen var relatert til produksjonsmodusen i React. Nå som appen faktisk ble kjørt i produksjon, fikk den ikke tilgang til Keycloak på samme måte lenger. Vi begynte derfor først å se om feilen kunne løses med å forandre nginx.conf, og brukte mye tid på å justere variabler i Docker-compose-fila og teste ulike løsninger.

Etter mye debugging fant vi ut at problemet handlet om at brukeren ble sendt til en URL som så sånn ut: <HTTP://auth:8080/>..., altså et internt Docker-vertsnavn som ikke er tilgjengelig fra eksterne nettlesere. Keycloak genererte omdirigeringsadresser basert på sitt interne navn i stedet for det offentlige domenet. Selv om frontend var riktig konfigurert med REACT_APP_KEYCLOAK_URL=HTTPs://auth.dev.maskinistene.no, måtte vi manuelt angi hvilken «Frontend-URL» Keycloak skulle bruke til omdirigering. Etter dette fungerte omdirireringene som de skulle, og login/registrering ble mulig igjen i staging.

Betalingssystemet:

Vi jobbet med å få på plass enhetstesting for backend/main. Det lå allerede noen tester der fra før, men etter at vi gjorde større endringer i koden, sluttet de fleste å fungere. Vi måtte derfor rydde opp og skrive nye tester som passet med den oppdaterte logikken.

Etter å ha jobbet i separate feature-grener i over en måned, slo vi sammen grenene. Vi begynte å koble sammen hele flyten fra frontend → backend → PDF. Det første vi startet på var en løsning som gjør det mulig å laste ned en PDF fra frontend, knyttet til en bestemt faktura. Det var en viktig del for å få hele systemet til å henge sammen i praksis.

I tillegg fikk vi integrert støtte for API_tokens og implementerte rate limiting ved hjelp av FastAPI sitt “SlowAPI”-bibliotek. Det begrenser hvor mange kall som kan gjøres til API-endepunktene per IP-adresse, og fungerer som et ekstra lag med beskyttelse mot misbruk.

Vi forberedte en code review og justerte rate limiteren slik at den kun begrenser uautoriserte brukere - altså de som ikke sender med API-token. Dette gjør at autoriserte systemer fortsatt får tilgang uten å bli bremset, mens vi samtidig beskytter oss mot potensielt misbruk fra anonyme kall.

Videre fullførte vi en funksjon som gjør det mulig å laste ned PDF-er direkte fra admin-panelet. Dette var mest for å teste at hele systemet fungerer sammen, mer enn at det var en kritisk funksjon. PDF-en henter fakturadata fra databasen, og vi har lagt inn testdata for medlemsinformasjon siden vi foreløpig ikke har koblet systemet til medlemsdatabasen. Utseendet er gjort klart for å ligne på hvordan det kommer til å se ut når den faktiske integrasjonen er på plass. Informasjonen om leverandøren («supplier») hentes via API-token, som slår opp data i databasen og legger det inn i PDF-filen. Vi skrev også ekstra enhetstester for backend og PDF-genereringen for å sikre at alt fungerer som forventet.

2.4.10 Sprint 10 – Klargjøring for produksjon:

Medlemskapssystemet:

Denne uka har vi jobbet videre med forberedelser til produksjon og generell opprydding i infrastrukturen. Vi oppdaterte nginx.conf slik at den nå inkluderer oppsett for server-auth, riktige headers og logging for å verifisere at headerne faktisk blir sendt korrekt.

Etter å ha gjennomgått Docker logs, fant vi ut at de eneste reelle feilene kom fra auth-containeren, altså Keycloak. Vi endret noen variabler som ga «deprecation-warnings» - altså advarsler om at de kommer til å forsvinne i fremtidige versjoner av Keycloak.

Vi oppdaterte prosjektets avhengigheter ved å gå gjennom og forandre versjoner i «package.JSON». Deretter oppdatere «package-lock.JSON» med de samme avhengighetene ved å kjøre «npm update». Denne filen sørger for at alle bruker de samme versjonene av pakkene, både i utvikling og produksjon. Disse avhengighets oppdateringene er viktig for å redusere risikoen for sikkerhetshull og forbedre ytelsen. Vi måtte også forsikre oss om at oppdateringene ikke introduserte nye feil, dette gjorde vi ved å teste systemet etter hver oppdatering.

Som en del av oppsettet for Keycloak la vi til en ny miljøvariabel i Docker-compose-filen, sammen med en «.JSON»-fil som definerer et ferdig konfigurert realm. Dette gjør at en Keycloak-realm automatisk opprettes ved oppstart av systemet, basert på innholdet i denne filen. Importeringen skjer kun dersom realmen ikke allerede finnes, noe som betyr at eventuelle manuelle endringer i et eksisterende realm ikke blir overskrevet ved oppstart.

En ting vi prøvde å få til, men ikke helt fikk på plass, var å få vårt eget custom tema i Keycloak til å automatisk bli valgt som standard. Det viste seg å være mer komplisert enn vi trodde. Vi kommer kanskje tilbake til det senere, men akkurat nå er det ikke høyeste prioriteten.

Fokuset vårt nå er å få systemet ut i produksjon, og samtidig begynne å prioritere å flytte Vipps-integrasjonen over i det nye betalingssystemet. Vi har også tenkt å rydde opp i miljøene våre, slik at staging og dev får egne URL-er. Det er fortsatt ikke påbegynt, men det står på lista.

Betalingssystemet:

Vi jobbet med å sette opp database-backup ved å ta utgangspunkt i løsningen oppdragsgiver tidligere hadde brukt i medlemskapssystemet. Den ble tilpasset vårt prosjekt og koblet opp med nødvendige endringer.

Det var en del utfordringer underveis, spesielt med å sette seg inn i hvordan scriptet til oppdragsgiver fungerte. Når programmet krasjet, var feilmeldingene ofte lite hjelpsomme. I tillegg måtte vi bruke dos2unix hver gang vi endret på filene, for å sikre riktig formatering. En annen bug dukket også opp miljø-variablene ble tolket med ekstra "" rundt seg, noe som førte til at scriptet ikke fungerte som det skulle. Vi fant aldri helt ut hvorfor det skjedde, men klarte å få scriptet til å ignorere den ekstra "" og kjøre som forventet.

Vi fikk lagt til flere nye tester blant annet rendering, template-prosessering og konvertering fra HTML til PDF. Dette ga oss bedre kontroll på kvaliteten og gjorde oss bedre rustet for produksjonssetting.

Vi hadde også et møte med oppdragsgiver der vi diskuterte veien videre for betalingssystemet. Etter møtet lagde vi noen diagrammer og fikk en bedre forståelse over hvordan Vipps burde integreres inn i systemet. Vi kom frem til at vi ikke skulle flytte over det eksisterende systemet fra medlemskap, men heller bygge vårt eget fra bunnen, bedre tilpasset vårt oppsett.

2.5 Kravspesifikasjonen og dens rolle

2.5.1 Endringer i kravspesifikasjonen

I dette prosjektet ble en del krav definert på forhånd, særlig for medlemskapssystemet, mens andre krav, særlig knyttet til betalingssystemet – vokste frem underveis. Dette skyldes både praktisk arbeid, tilbakemeldinger og gjennomganger sammen med oppdragsgiver, og behov som først ble synlig etter deler av systemet var implementert. Dette beskriver en iterativ og inkrementell utviklingsprosess.

Et eksempel på hvordan krav vokste fram underveis, var behovet for API-tokens og rate limiting i PDF-modulen. Etter hvert som løsningen tok form, ble det tydelig at API-endepunktet ikke kunne være åpent, da det skulle kjøres på oppdragsgivers domene, og ellers ville vært sårbart for misbruk. Derfor måtte vi implementere rate limiting i API-systemet, og tokens ble innført for å sikre autentisering.

Samtidig var det et opprinnelig krav at PDF-modulen skulle være databaseuavhengig og kunne brukes av hvem som helst via API-et. Innføringen av tokens utfordret dette kravet, siden autentiseringsnøkler måtte lagres et sted.

2.5.2 Kravspesifikasjonens rolle i design og implementering

For medlemskapssystemet var det utarbeidet en todo-liste av oppdragsgiver som fungerte som utgangspunkt for kravspesifikasjonen. Listen var delt inn i to deler: “Must have before official launch” og “Short term tasks”. Prioriteringene ble naturlig nok å ferdigstille oppgavene i “Must have before official launch”, og deretter se på oppgavene i “Short term tasks” dersom det var tid og kapasitet igjen. I tillegg ble **GitHub-oppgavelisten (issue tracker)** brukt som et praktisk verktøy for å samle, oppdatere og prioritere kravene.

Selv om kravspesifikasjonen ikke var fullstendig ved prosjektstart, spilte den en sentral rolle i utviklingsarbeidet. Den fungerte som en dynamisk backlog, der nye krav og justeringer ble lagt til underveis basert på praktiske erfaringer, tekniske avklaringer og tilbakemeldinger fra oppdragsgiver.

Kravene påvirket flere sentrale valg rundt design og implementering. Et tydelig eksempel er beslutningen om å bruke Mathesar som administrasjonsverktøy for medlemsdatabasen. Dette valget ble styrt av kravet om enkel administrasjon, altså lav terskel for bruk, men også minimal utviklingsinnsats.

2.5.3 Samsvar mellom krav og løsning

Stor sett samsvarer løsningene med kravene som ble definert i løpet av prosjektet. Alle krav som var merket med «må» eller «must have» i den opprinnelige todolisten ble implementert. Noen krav ble justert for å tilpasse seg nye behov eller tekniske begrensninger, særlig innen sikkerhet og API-håndtering.

Krav med lavere prioritet («bør») ble håndtert dersom det var kapasitet, og enkelte av disse oppstod også først i senere faser. Den iterative tilnærmingen gjorde det mulig å tilpasse utviklingen underveis, uten at det gikk ut over de viktigste kravene.

En full oversikt over

alle kravene kan leses under kapittel 3.3 Kravspesifikasjon i produktdokumentasjonen.

Listen med alle definerte krav kan leses under kapittel 3.3 Kravspesifikasjon i produktdokumentasjonen.

3. Produktdokumentasjon

3.1 Innledning

Vi har som beskrevet jobbet på to forskjellige systemer som vi har hatt som ambisjon å fullintegrere. Vi har vært fire studenter som har jobbet to og to på henholdsvis medlemskapssystemet og betalingssystemet. Under produktdokumentasjon har vi valgt å kjøre en tydeligere splitt på de forskjellige modulene, hvor vi først gjennomgår og dokumenterer medlemskapssystemet, for så gjøre det tilsvarende for betalingssystemet. Vi har funnet frem til at dette er den beste strukturen for dette kapittelet.

3.2 Systemarkitektur

Her gjennomgås den overordnede strukturen i prosjektet vårt og hvordan det er tenkt at betalingssystemet skal fungere sammen med medlemskapssystemet eller andre eventuelle aktører som tar det i bruk.

Medlemskapssystemet har allerede en integrert, men lettintelligens betalingsløsning. Denne betalingsløsningen er tett bygd inn til medlemskapssystemet, uoversiktig og vanskelig å bygge på. Oppdragsgiver ønsker at betalingsfunksjonalitet skal fjernes fra medlemskapssystemet og delegeres til et eget betalingssystem. Betalingssystemet skal være sin egen entitet, separat fra medlemskapssystemet med egen database og git-repository. Å dele systemet i to selvstendige systemer ville gjøre systemene mye mer oversiktlig og videreutvikling blir gjort mye lettere.

Betalingssystemet skal håndtere alt som har med betalinger å gjøre og medlemskapssystemet skal håndtere alt som har med medlemmer å gjøre. Betalingssystemet skal motta betalinger fra banken, oppdatere databasen sin og gi beskjed til medlemskapssystemet om betalte kundeforendringer. Betalingssystemet skal ikke vite noen ting av medlemsinformasjon og det er medlemskapssystemets jobb å styre hvilke utestående beløp og betalinger som er tilordnet hver kunde.

Sluttmålet er at betalingssystemet skal være åpen kildekode og kunne bli tatt i bruk av et hvilket som helst system som trenger en betalingsløsning. Betalingssystemet skal støtte mange ulike typer transaksjoner, men abstrahere "svaret" og gjøre det lett tolkelig for medlemskapssystemet som tar det i bruk. Betalingssystemet skal være et slags mellomledd mellom ulike betalingsverktøy/moduler og systemet som ønsker å ta det i bruk. Systemet skal

lagre betalinger og kundefordringer i databasen så det ikke er relevant hva slags betalingsmetode som er brukt.

Som en del av prosjektet vårt har vi også laget vårt eget faktura-modul, den såkalte PDF-modulen. En selvstendig modul som ikke er en del av betalingssystemet, men brukes av det. Modulen lager en PDF-faktura ut ifra en xml-oppskrift og tillater organisasjoner å legge inn sine detaljer og lage sitt eget faktura format med logo og nødvendige organisasjonsdetaljer (betalingsdetaljer, tlf, osv..).

3.2.1 Teknologistabel (Tech Stack)

En Teknologistabel (engelsk: «Tech Stack») er en betegnelse for å beskrive hvilken «gruppe» av teknologier, rammeverk, språk, teknologier, databaseverktøy, eller bedre sagt sett med verktøy man bruker for sin applikasjon (heap, u.å). Ofte er teknologistablene etablerte industristandarder, ofte knyttet opp mot forskjellige programmeringsspråk og deres tilhørende kompatible løsninger. Eksempler på stacks kan være .NET (Microsoft og C#), Spring (Java), Laravel (PHP). Flere av disse stackene har tilhørende biblioteker og løsninger for å håndtere backend, frontend, APIer, database ol.(heap, u.å).

Da vi begynte på dette prosjektet så var medlemskapssystemet allerede utviklet i Python. Dette er et språk vi har blitt introdusert til gjennom et tidligere fag. Det er et språk som er kjent for å være veldig fleksibelt, og som har mange interne og eksterne biblioteker, med gode teknologistabler for webapplikasjoner.

I betalingssystemet valgte vi å fortsette med Python som språk, og landet på FastAPI-stacken. Det er et godt og effektivt rammeverk for å bygge webapplikasjoner og APIer (FastAPI, u.å). Mye av grunnen til at vi brukte denne stacken er at den benytter og baserer seg på anerkjente Starlette-rammeverket for webapplikasjoner og Pydantic-bibliotekene som gjør det enkelt å sette opp PDF-modulen pga de gode kapabilitetene til å håndtere datatyper og validering (FastAPI, u.å). Vi ble også anbefalt å bruke denne teknologistabenen gjennom en av veilederne hos oppdragsgiver.

3.3 Kravspesifikasjon

Kravspesifikasjonen har vært et viktig verktøy for å styre utviklingen og vurdere resultatet av det endelige produktet. Ved prosjektoppstart var det foreslått en rekke definerte krav for medlemskapssystemet. Dette gjorde det nødvendig å prioritere hvilke som var mest kritiske for å oppnå hovedmålene. Enkelte krav ble utsatt eller nedprioritert for å sikre fremdrift og fokusere på helheten. For betalingssystemet utviklet kravene seg i større grad underveis. Behov og funksjonalitet ble justert basert på våre praktiske erfaringer med systemet, tilbakemeldinger fra oppdragsgiver og tekniske avklaringer.

Kravene har vi delt inn i funksjonelle og ikke funksjonelle krav. De funksjonelle kravene beskriver hva systemene skal gjøre og beskriver de funksjonene som systemene skal kunne utføre eller tilby. De ikke funksjonelle kravene beskriver egenskaper ved systemene. Dette er krav som stiller begrensninger til hvordan systemene skal fungere, og omhandler systemenes kvalitet, ytelse, sikkerhet, tilgjengelighet og brukervennlighet (Sommerville, 2016).

Tabellene nedenfor viser en oversikt over de funksjonelle og ikke funksjonelle kravene som gjelder for hver modul, prioritet, hvorvidt kravet ble oppfylt og en kort beskrivelse av implementasjonen av de funksjonelle kravene.

3.3.1 Funksjonelle krav

Medlemskapssystem

Krav	Prioritet	Oppfylt	Implementasjon
Skinning av Keycloak tema	Må	Ja	Keycloak er konfigurert med et egendefinert tema som inkluderer EFN's farger, logo og typografi. Temaet er aktivert og synlig i login- og registreringsprosessen til medlemskapssystemet.
Enkel administrering av medlemmer	Må	Ja	Brukergrensesnittet Mathesar er integrert for å la administratorer enkelt lese, redigere og slette medlemsdata direkte i databasen uten teknisk kompetanse.

Privacy-statement i frontend	Må	Ja	En personvernerklæring er implementert og tilgjengelig via en klikkbar lenke i frontend ved registrering av nytt medlem. Når brukeren klikker på lenken, vises den fullstendige erklæringen.
Systemet skal sende bekreftelsesmail til nye betalte medlemmer	Bør	Delvis	Epost utsending satt opp via SMTP

Betalingsystem

Krav	Prioritet	Oppfylt	Implementasjon
Håndtere og verifisere innbetalinger	Må	Ja	Utviklet et API med verifiseringsmodul og database for betalingsstatus.
Automatisk generering av faktura (PDF)	Må	Ja	Fakturamodul bygget «fra bunnen». Bruker XML som input.
Admin-oversikt for ubetalte fakturaer	Må	Ja	Implementert en frontend-modul for å markere faktura som betalt. Behov oppdaget under testing.

Systemet skal inneholde en database som lagrer informasjon om utestående og gjennomførte betalinger.	Må	Ja	Det er tatt i bruk en postgresql database som holder styr på dette.
PDF-modulen skal generere fakturaer i PDF-format med et profesjonelt utseende.	Må	Ja	En HTML-mal er benyttet for å sikre korrekt struktur og oppbygging av genererte PDF-dokumenter.
Betalingssystemet skal støtte manuell registrering av kontantbetalinger.	Bør	Ja	Det er implementert en funksjon i administrasjonspanelet som gjør det mulig å opprette nye betalinger og markere dem som betalt manuelt.
Betalingssystemet skal være integrert med medlemskapssystemet for å knytte betalinger til gyldige medlemskap.	Må	Nei	Ikke påbegynt
Systemet skal benytte webhooks for å kunne motta og behandle betalingsbekreftelser fra banker automatisk.	Må	Nei	Ikke påbegynt
Systemet skal støtte integrasjon med Vipps som betalingsmetode.	Hvis tid	Nei	Ikke påbegynt
Systemet skal støtte betaling med betalingskort (f.eks. Visa, Mastercard)	Hvis tid	Nei	Ikke påbegynt

Systemet skal støtte betaling med bitcoin	Hvis tid	Nei	Ikke påbegynt
---	----------	-----	---------------

3.3.2 Ikke-funksjonelle krav

Medlemskapssystem

Krav	Prioritet	Oppfylt?
Systemets avhengigheter skal være oppdatert, inkludert de som ikke håndteres av automatiske verktøy.	Må	Ja
Systemet skal sikre at det tas regelmessige og pålitelige sikkerhetskopier av databasene.	Må	Ja
Gjenopprettingsprosedyren skal være enkel, testbar og dokumentert.	Må	Ja
Frontend-komponenter skal bygges i produksjonsmodus uten debug-innstillinger.	Må	Ja
Feil i logger for Keycloak og databasen skal være håndtert.	Må	Ja
Systemet skal kunne testes i staging-miljøet.	Må	Ja
Systemet skal ikke ha kjente feil i login og registrering i staging-miljøet.	Må	Ja

Betalingssystem

Krav	Prioritet	Oppfylt?
Reentrant og token-basert tilgang	Bør	Ja
All kildekode skal være dekket av enhetstester.	Må	Ja
Systemet skal kunne distribueres og kjøres i et Docker-miljø	Må	Ja
Systemet skal med riktig oppsett også kunne kjøres uten Docker.	Bør	Ja
Systemet skal være plattformuavhengig og kunne startes korrekt uavhengig av operativsystem og maskinvare.	Må	Ja
PDF-systemet skal ha innebygd begrensning (rate limiting) som regulerer hvor mange instanser som kan kjøres samtidig.	Bør	Delvis
Systemet skal sikre at det tas regelmessige og pålitelige sikkerhetskopier av databasene.	Må	Delvis. Gjennomført i en feature branch, men ikke i main

3.4 Medlemskapssystemet

3.4.1 Forord

Dette kapittelet gir en detaljert teknisk gjennomgang av medlemskapssystemet slik det fungerer i sin helhet. Store deler av systemet eksisterte allerede før arbeidet med

bachelorprosjektet startet, derfor er det like viktig å gå gjennom eksisterende funksjonalitet som det er å gå gjennom nyutviklet funksjonalitet. Målet er å gi en komplett oversikt over systemets oppbygning, samspill mellom komponenter og systemets funksjoner, slik at dokumentasjonen også kan brukes for eventuelle videreutviklinger av systemet.

Det er viktig å nevne selv om vi har gjort mye arbeid i dette systemet og lært mye underveis, er det fortsatt deler av funksjonaliteten og oppbygningen vi ikke har full oversikt over.

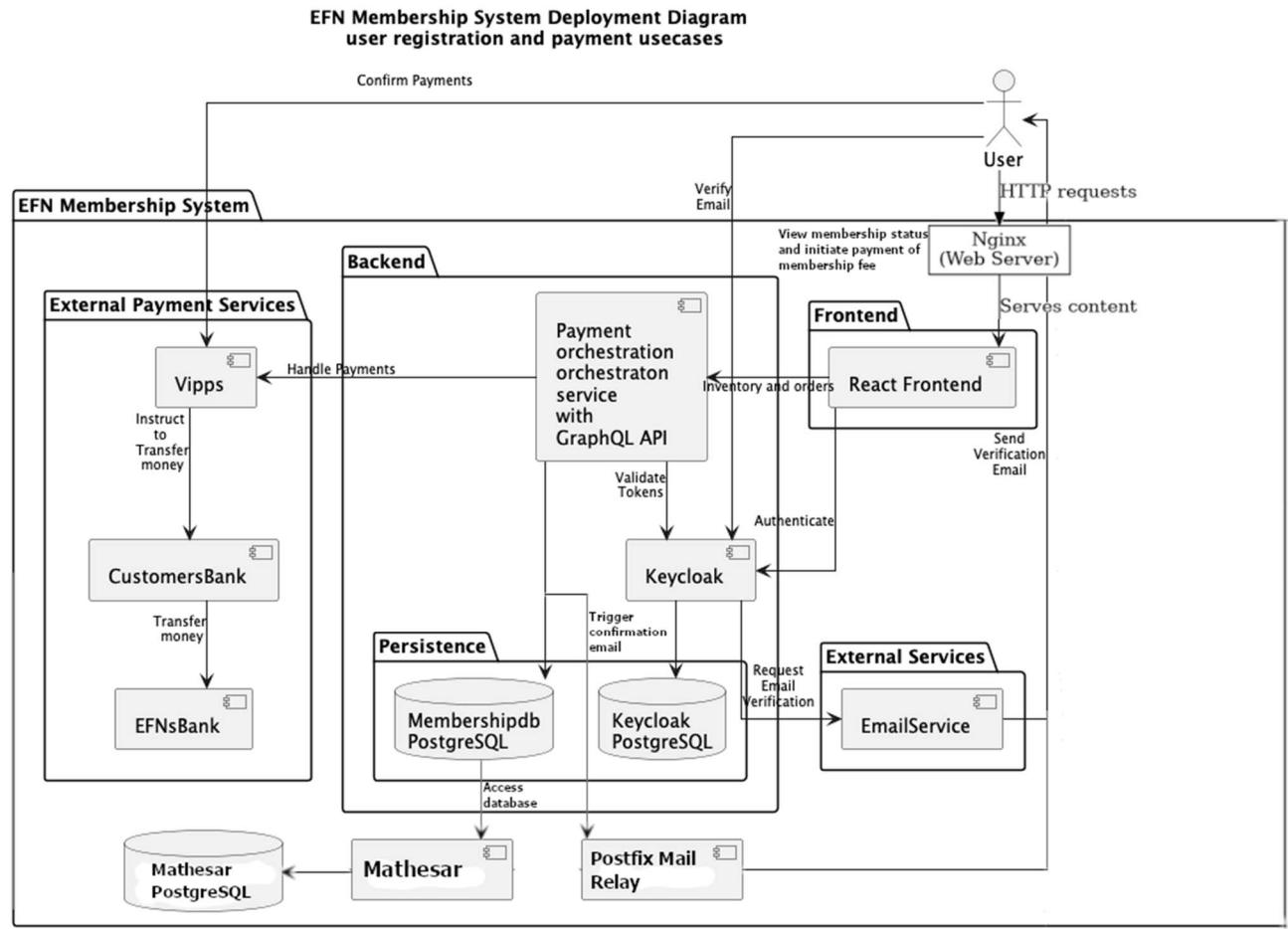
3.4.2 Oversikt over komponenter medlemskapssystem

Systemet består hovedsakelig av to kjernekategorier: frontend og backend, samt flere støttetjenester. Dette er illustrert i figur 5, som viser en oppdatert versjon av det opprinnelige systemdiagrammet laget av oppdragsgiver.

Frontend: er utviklet i React og er et grensesnitt der medlemmer kan registrere seg, logge inn, betale medlemskap og se sitt digitale medlemskort. React-applikasjonen kjøres som en egen Docker-container og tilgjengeliggjøres til brukeren via en Nginx-server.

Backend: utgjør kjernen i systemets funksjonalitet og håndterer blant annet autentisering og behandling av medlemsdata. Kommunikasjonen mellom frontend og backend baserer seg på en GraphQL API, som er en type API som lar klienten spesifisere nøyaktig hvilken data den trenger. Mye kjernefunksjonalitet er samlet i «orchestraion service», denne tjenesten overtar etter at brukeren er autentisert via Keycloak, og bruker ID-token for å hente og behandle data knyttet til medlemskap og betaling. For administrasjon og innsyn i medlemsdata ble Mathesar lagt til som et grafisk grensesnitt for PostgreSQL-databasene.

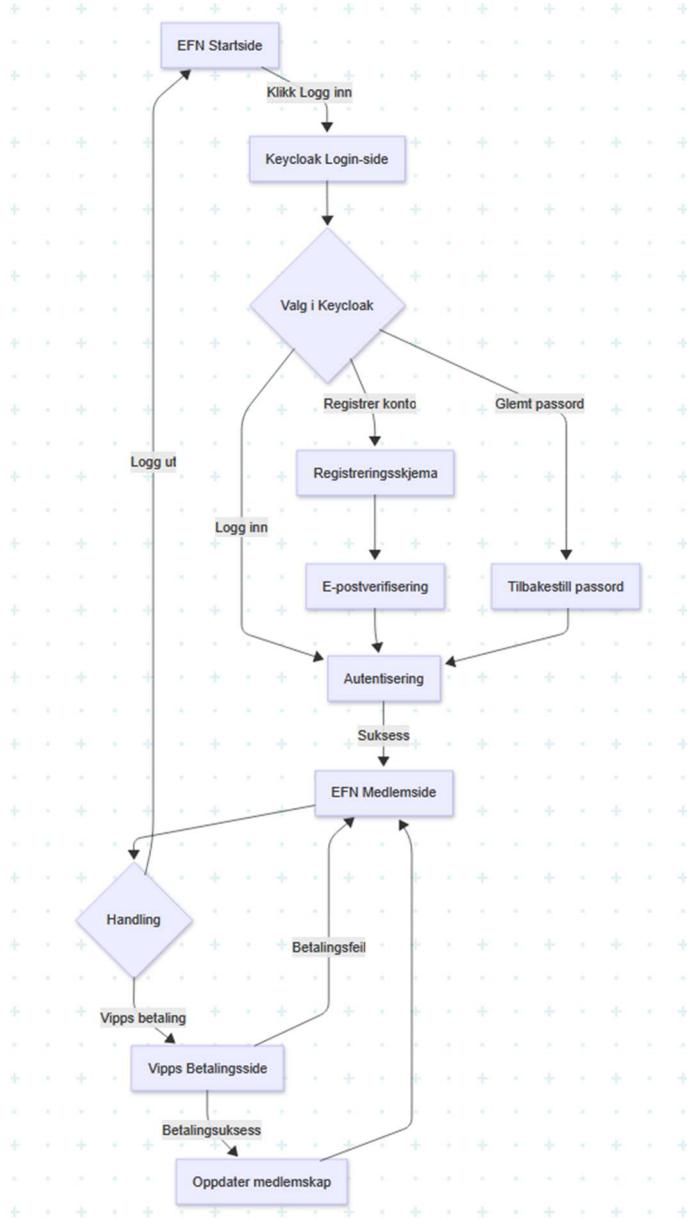
Om en bruker registerer seg på systemet vil «EmailService» sende en «Bekreft e-postadresse»-e-post til brukeren. I tillegg til denne ble det utviklet en ny «Postfix Mail Relay» som brukes til å sende e-post bekrefteelse når en bruker betaler for et medlemskap. Denne aktiveres når et medlem blir lagt til i databasen for betalte medlem. Systemet består av tre PostgreSQL-databaser, en for betalte medlemmer, en for autentiseringsssystemet Keycloak og en for Mathesar.



Figur 5 Diagram fra EFN, redigert for å passe til å vise delene vi har lagt til

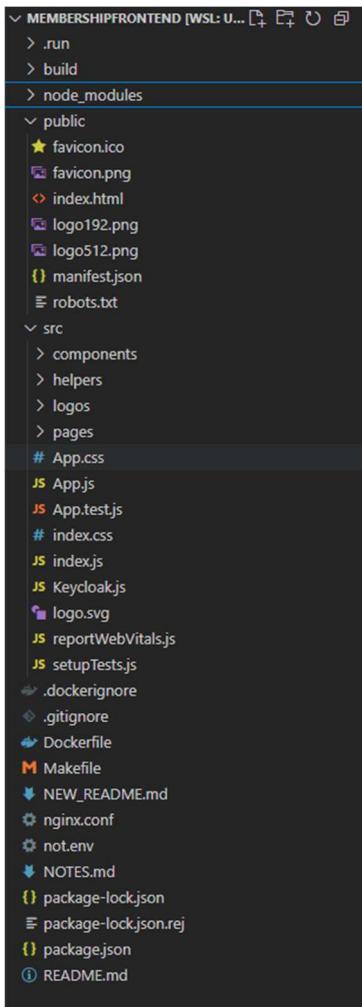
3.4.3 Brukerflyt

Løsningen er utviklet for å tilby en brukervennlig medlemsportal, der brukere kan logge inn, forandre passord og betale via Vipps. Autentisering og kontohåndtering skjer gjennom Keycloak. Diagrammet under (se figur 6) gir en oversikt over arkitekturen slik den var definert fra starten, strukturen forble uendret gjennom hele utviklingsprosessen.



Figur 6 Brukerflyt i medlemsportalen

3.4.4 Frontend



Figur 7 Mappestrukturen i frontend-applikasjonen

Frontend-delen av systemet er bygget med React og alt som har med frontend ligger i mappen «membershipfrontend». Koden følger en struktur der ulike deler av applikasjonene er plassert i egne mapper basert på funksjonalitet. Dette gjør det lettere å forstå hvordan frontend er bygget opp, og gjør det enklere å orientere seg i hvilke filer som tilhører hvilken del. Mappen «public» inneholder statiske ressurser som ikoner, manifestfiler og selve HTML-malen som React bygger på. Den faktiske programlogikken ligger i «src»-mappen, som består av flere undermapper.

For å gi applikasjonen et mer profesjonelt uttrykk, erstattet vi standard favicon med EFN sitt eget ikon. Dette ble plassert i «public»-mappen sammen med de andre statiske ressursene.

Vi gjorde noen mindre justeringer i frontend-koden for å klargjøre systemet for produksjon og forbedre vedlikeholdbarheten. For å kunne kjøre frontenden som en fullverdig produksjonsløsning, la vi til en eksplisitt «npm run build»-kommando. Denne bygger en optimalisert statisk versjon av React-applikasjonen.

For å levere de statiske filene valgte vi å bruke nginx - en effektiv og lettvekts webserver som er godt egnet til å håndtere statiske filer. For å ha kontroll over hvordan frontend-applikasjonen blir levert, opprettet vi en egen «nginx.conf»-fil som ble brukt i Docker-oppsettet vårt (se figur 9). Denne filen definerer hvordan Nginx skal håndtere forespørsler og caching.

I «nginx.conf»-filen satt vi opp to serverkonfigurasjoner, en på port 3000 for frontend-applikasjonen som inkluderer regler for caching av statiske ressurser som bilder, JavaScript og CSS filer. Her sørger vi også for at hovedsiden ikke caches, slik at brukeren alltid får den nyeste versjonen. Den andre serverkonfigurasjonen er på port 80 og håndterer forespørsler til Keycloak's autentiseringstjeneste. Denne videresender trafikken direkte til Keycloak, som sørger for at riktige HTTP-headere blir sendt og at Keycloak kan behandle forespørslene korrekt.

For å unngå feil som «missing clientID» måtte vi sørge for at nødvendige miljøvariabler ble tilgjengelige under selve byggingen av frontend-applikasjonen. Vi brukte byggargumentet «ARG» i «Dockerfile»-filen for å sende inn verdiene under byggeprosessen (se figur 8). Disse verdiene blir da hardkodet inn i den ferdig bygde applikasjonen, som da sikrer at all nødvendig informasjon om autentisering og API-kommunikasjon er tilgjengelig allerede ved oppstart.

Vi brukte en multi-stage Docker-build for å skille bygging og kjøring av frontend-applikasjonen (se figur 8). Først bygges applikasjonen i et Node.js-basert miljø, deretter overføres de kompilerte filene til nginx. Til levering av de statiske filene valgte vi å bruke «nginx:1.27-alpine» for å sikre kompatibilitet med basen for byggefase som er «node:22-alpine». Begge disse bildene er basert på «Alpine Linux 3.21.3», noe som skal sikre god kompatibilitet mellom byggefase og produksjonsmiljøet.

```

1  # Create image based on the official Node image from dockerhub
2  # Using hints from here: https://www.docker.com/blog/9-tips-for-containerizing-your-node-js-application/
3  FROM node:22-alpine AS build
4
5  # Pass build-time variables (to fix missing clientid)
6  ARG REACT_APP_GRAPHQL_URL
7  ARG REACT_APP_KEYCLOAK_URL
8  ARG REACT_APP_KEYCLOAK_REALM
9  ARG REACT_APP_KEYCLOAK_CLIENT_ID
10
11 # Set the environment to production
12 ENV NODE_ENV=production
13
14 # Create app directory
15 WORKDIR /usr/src/app
16
17 # Copy dependency definitions
18 COPY package.json ./package.json
19 COPY package-lock.json ./package-lock.json
20 |
21 RUN npm ci
22
23 # Get all the code needed to run the app
24 COPY ..
25
26 RUN npm run build
27
28 # Using nginx:1.27-alpine to ensure compatibility with node:22-alpine,
29 # as both images are based on Alpine 3.21.3.
30 FROM nginx:1.27-alpine
31
32
33 COPY --from=build /usr/src/app/build /usr/share/nginx/html
34 COPY nginx.conf /etc/nginx/conf.d/default.conf
35
36 EXPOSE 3000
37 CMD ["nginx", "-g", "daemon off;"]

```

Figur 8 Dockerfile i membershipfrontend

```

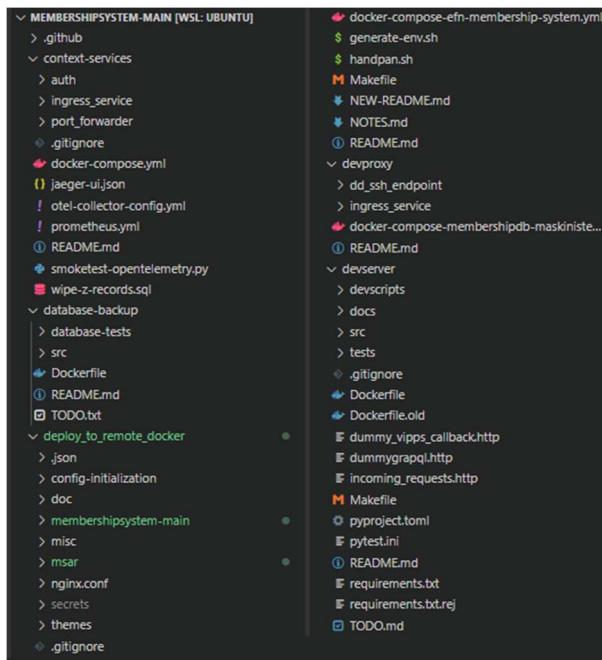
1 server {
2     listen 3000; # Sets nginx to listen on port 3000
3     server_name _;
4
5     # Add Log
6     access_log /dev/stdout combined;
7     error_log /dev/stderr;
8
9     # Serve static files with proper caching
10    location / {
11        root /usr/share/nginx/html; # Where website files are stored
12        index index.html index.htm; # Default files to serve
13        try_files $uri /index.html; # For SPAs: if file not found, serve index.html
14
15        # Prevents browsers from storing the main page in cache
16        add_header Cache-Control "no-cache, no-store, must-revalidate";
17    }
18
19    # Special rules for image and asset files
20    location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {
21        root /usr/share/nginx/html; # Same folder as above
22        expires 30d; # Keep in browser cache for 30 days
23        add_header Cache-Control "public, max-age=2592000"; # 30 days in seconds
24    }
25 }
26 server {
27     listen 80; # Listen for requests on port 80
28     server_name auth.dev.maskinistene.no; # This server block handles requests for this domain
29
30     # Add Log
31     access_log /dev/stdout combined;
32     error_log /dev/stderr;
33
34     location / {
35         proxy_pass http://auth:8080/; # Forward the request to Keycloak's internal service
36
37         # Forward headers using the standardized Forwarded header
38         proxy_set_header Host $host;
39         proxy_set_header Forwarded "for=$remote_addr;host=$host;proto=$scheme";
40     }
41 }

```

Figure 9 nginx.conf

3.4.5 Backend

Backend-delen av systemet består av flere tjenester og skript som kjører sammen for å muliggjøre autentisering, e-postutsending og generell systemlogikk. Selv om mye av infrastrukturen allerede var på plass har vi gjort betydelige forbedringer og tilpasninger til systemet. Figur 10 viser nesten hele mappestrukturen for prosjektet, og gir et overblikk over hvordan backend-delen er organisert.



Figur 10 Mappestrukturen som viser hvordan backend-delen systemet er satt opp

En av de første tingene vi tok tak i var å oppdatere tredjepartsavhengighetene i «devserver», som utgjør kjernen av backend-logikken i systemet. Her håndteres blant annet oppsett for GraphQL, API-forespørsler og integrasjoner mot databasen. Vi gjorde oppdateringene systematisk ved å følge GitHub Dependabots «pull requests» (se figur 11), som varsler om kjente sårbarheter i eldre avhengighet-versjoner. Dette ble gjort for å lukke potensielle sikkerhetshull og sikre at vi har et mer robust system.

Tornado has an HTTP cookie parsing DoS vulnerability #68

Fixed Opened 6 months ago on [tornado \(pip\)](#) · [devserver/requirements.txt](#) · Fixed 4 months ago

Package	Affected versions	Patched version
tornado (pip)	<= 6.4.1	6.4.2

The algorithm used for parsing HTTP cookies in Tornado versions prior to 6.4.2 sometimes has quadratic complexity, leading to excessive CPU consumption when parsing maliciously-crafted cookie headers. This parsing occurs in the event loop thread and may block the processing of other requests.

See also [CVE-2024-7592](#) for a similar vulnerability in cpython.

dependabot bot opened this 6 months ago

dependabot bot closed this as completed in #64 4 months ago

Figur 11 Et eksempel på en lukket "pull request" fra Dependabot

Fordi dette systemet skal være åpen kildekode måtte alt som blir sett på som hemmeligheter eller nærmere bestemt sensitiv data bli håndtert på en forsvarlig måte. Man kan ikke ha sensitiv data som passord og tilgangsnøkler lagt direkte i kildekoden. Dette ble løst ved å bruke en generator-basert tilnærming der miljøspesifikke filer automatisk opprettes ved å bruke data fra .sh-filer. Genereringen skjer gjennom et generate.env-skript og en samling .sh-filer som ligger i secrets/basis, alt i secrets mappen ligger utenfor versjonskontrolleren. Hver .sh-fil inneholder hemmelige variabler knyttet til et bestemt miljø, altså miljøer som «local», «development», «staging» og «production». Disse filene behandles videre av et «Makefile»-system som gjør det enkelt å generere de nødvendige «.env», «.yml» og kjørbare «Docker-compose.sh»-filer.

For å gjøre det enkelt for andre å sette opp systemet, laget vi en «DEPLOYMENT.md»-fil (se figur 12). Denne beskriver steg for steg hvordan systemet kan kjøres opp både lokalt og på ekstern server i forskjellige miljøer (dev, staging og prod). Den forklarer hvordan man konfigurerer miljøspesifikke variabler, genererer nødvendige filer og starter opp systemet i ønsket miljø. Dokumentet inneholder også forklaringer for hvordan man kobler seg til Keycloak via en sikker SOCKS5-proxy, samt hvordan man kan importere en forhåndsdefinert realm-konfigurasjon. Fordi dokumentet skal brukes av andre til å sette opp systemet var det viktig at vi testet det, måten vi gjorde dette på var ved å gå gjennom dokumentet fra start til slutt og bare følge det som stod der. Om det ikke var tilstrekkelig for å sette opp systemet visste vi at vi måtte forandre det.

Deployment guide

This guide helps you deploy the membership system to various environments: local, staging, and production using remote Docker instances.

Prerequisites

Ensure you have the necessary files available in the secrets directory. Only the files in secrets/basis need to be provided by the developer. These files are .sh scripts that declare environment-specific variables.

Directory Structure

- secrets/basis: Contains shell scripts (.sh) that define environment variables.
- secrets/configs: Contains .yml files for Docker Compose.
- secrets/env: Contains .env files for Docker Compose.
- secrets/bin: Contains scripts for starting and stopping the deployment.

Environment Setup

Local Deployment

Prepare Environment Variables Create or modify `secrets/basis/dev-localhost.sh` with the necessary variables for local deployment (see example below).

```
SCENARIO_NAME="development localhost"
COMPOSE_PROJECT_NAME="test_localhost"
BACKDOOR_EXPOSED_SSL_PORT=223
AUTH_ADMIN_PASSWORD=
DATABASES_BASE_DIRECTORY="/tmp/membershipsystem-test-database"
SCENARIO_TYPE="development"
VIPIPS_CLIENT_ID=
VIPIPS_CLIENT_SECRET=
VIPIPS_MERCHANT_SERIAL_NUMBER=""
VIPIPS_SUBSCRIPTION_KEY=
AUTH_DATABASE_PASSWORD="example_password"
MBD_DATABASE_PASSWORD="example_password"
```

Generate Required Files

Run the following command to clean old files and generate new ones: `make clean all`

Deploy locally

Execute the deployment script:

```
secrets/bin/docker-compose-dev-localhost.sh up -d
```

Staging and Production Deployment

The process for staging and production is similar to local deployment. Ensure the basis files (secrets/basis/staging-virtual2.sh and secrets/basis/prod-virtual2.sh) are properly configured with their respective environment variables.

Figur 12 Utsnitt av "DEPLOYMENT.md"-filen

I systemet benyttes Docker Compose for å definere oppstarten av de ulike tjenestene. Dette styres gjennom «Docker-compose-efn-membership-system.yml», den spesifiserer hvilke containere som skal kjøre, hvilke miljøvariabler og nettverk som skal brukes. Vi har gjort en rekke endringer i denne filen. Blant annet har vi integrert Mathesar som verktøy for databaseadministrasjon (se figur 13), lagt til støtte for e-posttjenesten Postfix og opprettet «healtcheck»-rutiner for databaser som må være i drift før avhengige tjenester starter. Vi endret flere miljøvariabler som var markert som «deprecated». Det betyr at de fortsatt virker, men skal fases ut og derfor bør erstattes.

```

mathesar:
  image: mathesar/mathesar:0.2.0
  environment:
    SECRET_KEY: ${MATHESAR_POSTGRES_SECRET_KEY}
    DOMAIN_NAME: ${DOMAIN_NAME:-http://localhost}/

    # Mathesar's own internal database
    POSTGRES_DB: ${MATHESAR_POSTGRES_DB_NAME}
    POSTGRES_USER: ${MATHESAR_POSTGRES_DB_USER}
    POSTGRES_PASSWORD: ${MATHESAR_POSTGRES_DB_PASSWORD}
    POSTGRES_HOST: mathesar_postgres
    POSTGRES_PORT: 5432

    DJANGO_SETTINGS_MODULE: config.settings.production
    ALLOWED_HOSTS: ${ALLOWED_HOSTS:-*}
    VIRTUAL_PORT: "8000"
    VIRTUAL_HOST: ${ADMIN_VIRTUAL_HOST}
    LETSENCRYPT_HOST: ${MEMBERSHIPDB_ADMIN_LETSENCRYPT_HOST}
  volumes:
    - ./msar/static:/code/static
    - ./msar/media:/code/media
  depends_on:
    mathesar_postgres:
      condition: service_healthy
  networks:
    - ${MAGIC_SSL_PROXY_NETWORK:-magicSSLProxy} # Use network if specified
    - default

```

Figur 13 Oppsettet for Mathesar i Docker-compose.yml



Figur 14 Filstrukturen til vårt egendefinert tema

Keycloak har støtte for bruk av ulike temaer, som gjør det mulig å tilpasse blant annet innloggingsskjema, admin-konsoll, registreringsskjema og velkomstsiden. Et tema består av HTML, CSS, bilder, meldingsfiler og «theme.properties». Sistnevnte er en fil som forteller Keycloak hva slags tema det er, og hvilken base det eventuelt bygger på. I vårt tilfelle bygger temaet på standardtemaet, og vi da kun overstyre enkle deler av utseende. Senere så skulle vi legge til ekstra tekst på registreringsskjemaet, da måtte vi opprette en egen «register.ftl»-fil (se figur 14). Dette er en Freemarker-mal som overstyrer det opprinnelige registreringsskjemaet, for å kunne legge til egendefinert tekst og lenke til

personvernerklæringen (se figur 15).

Ved å opprette en konto og registrere deg på vår tjeneste, gir du ditt samtykke til at vi samler inn og behandler personopplysninger som beskrevet i denne [personvernerklæringen](#).

Figur 15 Egendefinert tekst fra registerinsskjema

For å dokumentere og sikre kvaliteten på backup-løsningen, gjennomførte vi en rekke tester. Målet var å kontrollere at gjenopprettede data faktisk samsvarer med originalene. For å bekrefte at sikkerhetskopiene faktisk inneholdt riktig data, utviklet vi et Python-script med unittest-framework.

```

1 import unittest
2 import subprocess
3
4 # Configuration
5 CONTAINER_NAME = "test_localhost-mdb_postgres-1"
6 DB_USER = "membership"
7 DB_NAME = "membership"
8
9 def run_sql_command(query):
10     """Run a SQL command into the Docker container and return the output."""
11     cmd = f"docker exec -i {CONTAINER_NAME} psql -U {DB_USER} -d {DB_NAME} -tAc \\"{query}\\"
12     result = subprocess.run(cmd, shell=True, capture_output=True, text=True)
13     return result.stdout.strip()
14
15 class Testdatabase(unittest.TestCase):
16     def test_table_efnmembership_exists(self):
17         """Test if the table 'efnmembership' exists."""
18         table_exists = run_sql_command("SELECT to_regclass('public.efnmembership');")
19         self.assertEqual(table_exists, "", "The table 'efnmembership' does not exist!")
20
21     def test_table_orders_exists(self):
22         """Test if the table 'orders' exists."""
23         table_exists = run_sql_command("SELECT to_regclass('public.orders');")
24         self.assertEqual(table_exists, "", "The table 'orders' does not exist!")
25
26     def test_table_product_exists(self):
27         """Test if the table 'product' exists."""
28         table_exists = run_sql_command("SELECT to_regclass('public.product');")
29         self.assertEqual(table_exists, "", "The table 'product' does not exist!")
30
31     def test_table_productinstance_exists(self):
32         """Test if the table 'productinstance' exists."""
33         table_exists = run_sql_command("SELECT to_regclass('public.productinstance');")
34         self.assertEqual(table_exists, "", "The table 'productinstance' does not exist!")
35
36     def test_user_count(self):
37         """Check that there is the correct number of users in the efnmembership table."""
38         expected_user_count = 6 # Should match the backup
39         actual_user_count = int(run_sql_command("SELECT COUNT(*) FROM efnmembership;"))
40         self.assertEqual(actual_user_count, expected_user_count, f"Expected {expected_user_count} users, but found {actual_user_count}!")
41
42     def test_specific_user_exists(self):
43         """Check that a specific user exists."""
44         user_exists = int(run_sql_command("SELECT COUNT(*) FROM efnmembership WHERE id = '8e22ad0c-d627-4788-8458-3de5573fcfb4';"))
45         self.assertGreater(user_exists, 0, "The user with ID 8e22ad0c-d627-4788-8458-3de5573fcfb4 does not exist!")
46
47     def test_no_null_order_id(self):
48         """Check that there are no NULL values in order_id."""
49         null_order_count = int(run_sql_command("SELECT COUNT(*) FROM efnmembership WHERE order_id IS NULL;"))
50         self.assertEqual(null_order_count, 0, f"There are {null_order_count} rows with NULL order_id!")
51
52 if __name__ == "__main__":
53     unittest.main()
54

```

Figur 16 Testscript for validering av gjenopprettede backups

Det var et ønske fra oppdragsgiver om å kunne sende e-post til nye medlemmer etter at betalingen er vellykket. For å få dette til satte vi opp en SMTP-satellittløsning gjennom Postfix og koblet dette til et Python-program (se figur 17) som vi utviklet for å håndtere e-postutsendingen. I funksjonen «insert_membership()» i «postgres_interface.py» ble det lagt inn en linje som kaller «send templated_email()» etter at et nytt medlemskap er registrert. Funksjonen henter e-postadressen og sender en ferdig generert e-post basert på en mal.

Disse malene er laget for å støtte ulike typer e-poster, og inneholder tekst for både emne og innhold i selve e-posten (se figur 18). Python-program prøver først å hente mal fra et sted, om det ikke er noe her vil den gå for standardmalene. Vi valgte denne løsningen fordi når noen

andre enn EFN skal ta i bruk systemet så er det veldig enkelt for dem å kunne lage sine egne e-post-maler uten å måtte forandre noe av koden. Det er bare malen for bekrefstelse av medlemskap som brukes, men det er laget mal for «error»- og velkomst e-poster også, som enkelt kan aktiveres ved behov.

```

❶ send_email.py
1  import smtplib
2  import os
3  from email.mime.text import MIMEText
4  from email.mime.multipart import MIMEMultipart
5
6  def send_templated_email(recipient_email, email_type):
7      """
8          Send mail to member
9
10     Args:
11         recipient_email (str): The email address of the recipient
12         email_type (str): Type of email to send (membership_confirmation, error, greeting, etc. like "membership_confirmation")
13
14     Returns:
15         bool: True if email was sent successfully, False otherwise
16     """
17
18     # Check for user template first, fall back to default
19     user_tmpl = f"templates/userdefined/{email_type}.tmpl"
20     default_tmpl = f"templates/{email_type}.tmpl"
21
22     template_path = user_tmpl if os.path.exists(user_tmpl) else default_tmpl
23
24     try:
25         # Read the template file
26         with open(template_path, 'r') as file:
27             template_content = file.read()
28
29         # Store the components of the email
30         sender_email = os.environ.get('POSTFIX_always_sender')
31         subject = ""
32         body = ""
33
34         # Go through each line of the template looking for specific labeled sections
35         for line in template_content.split('\n'):
36             if line.startswith("SUBJECT:"):
37                 subject = line.replace("SUBJECT:", "").strip()
38             elif line.startswith("BODY:"):
39                 body_start = template_content.find("BODY:") + len("BODY:")
40                 body = template_content[body_start:].strip()
41                 break
42
43         # Create a multipart message
44         message = MIMEMultipart()
45         message["From"] = sender_email
46         message["To"] = recipient_email
47         message["Subject"] = subject
48
49         # Attach the body to the email
50         message.attach(MIMEText(body, "plain"))
51
52
53
54         # Connect to the local SMTP server (postfix container)
55         server = smtplib.SMTP('postfix', 25)
56
57         # Send the email
58         server.send_message(message)
59
60         # Close the connection
61         server.quit()
62
63         print(f"{email_type} email sent to {recipient_email}")
64         return True
65
66     except Exception as e:
67         print(f"Failed to send {email_type} email: {e}")
68         return False

```

Figur 17 Innholdet i «send_email.py», som håndterer e-postutsending

```

EXPLORER ...  

OPEN EDITORS ...  

membership_confirmation.tpl templates  

PERSISTENCE [WSL: UBUNTU] ...  

templates  

> userdefined  

error.tpl  

greeting.tpl  

membership_confirmation.tpl  

membership_confirmation.tpl X  

templates > membership_confirmation.tpl  

1 SUBJECT: EFN Membership Confirmation  

2 BODY:  

3 Your payment was successful. Thank you for your payment!  

4  

5 You are now a member of EFN.  

6  

7 Best regards,  

8 EFN Team

```

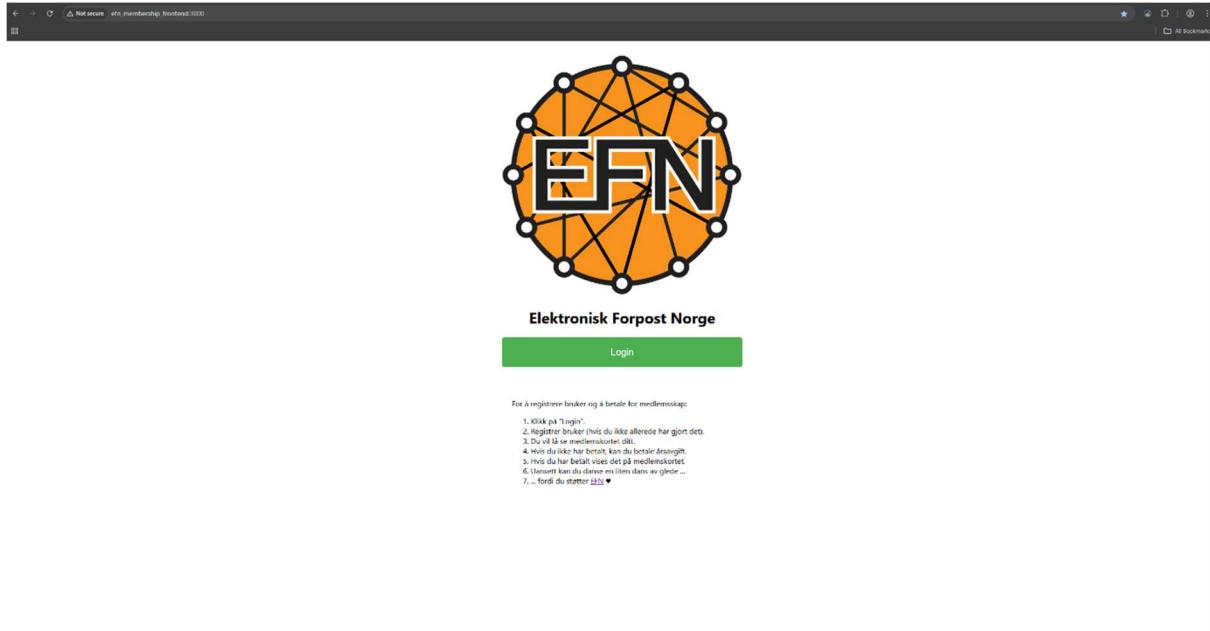
Figur 18 E-postmaler

3.4.6 Brukergrensesnitt (UI)

3.4.6.1 EFN startside (medlemsportal)

EFN sin startside er landingsside når en bruker trykker på «Bli medlem»-linken på efn.no.

Denne siden fungerer seinere som et digitalt medlemskort (se figur 20). Om en bruker er allerede innlogget vil de bli tatt rett til det digitale medlemskortet. For ikke-innloggede brukere vil man måtte trykke på «login»-knappen som tar dem videre til brukerinnlogging (se figur 19).



Figur 19 EFN startside

3.4.6.2 Digital medlemskort

Her kan man se om man er aktivt betalende medlem av EFN, om man ikke er det vil det bli vist en «Kjøp medlemskap» knapp.



Figur 20 Digital medlemskort

3.4.6.3 Brukerinnlogging

I brukerinnlogging benyttes Keycloak for autentisering, så her blir brukeren sendt videre til en URL som er konfigurerert som en del av Keycloak sin «redirect flow», dette sørger for at brukeren autentiseres direkte mot Keycloak sitt grensesnitt. Brukerinnloggingssiden gir brukeren mulighet for å logge inn, registrere bruker eller få tilbakestilt passordet sitt (se figur 21). Etter vellykket innlogging blir brukeren returnert til medlemsportalen (men nå vises det digitale medlemskortet), via en ID-token fra Keycloak hentes informasjon som brukernavn, e-post og medlemsstatus. Denne prosessen sikrer at kun verifiserte brukere får tilgang til medlemsspesifikke funksjoner, som visning av medlemskort og betaling av medlemskap.

Brukerinnloggingsiden er den siden det ble lagt mest vekt på at vi måtte lage et tema til, fordi standardtemaet skilte seg ut og passet ikke visuelt med resten av EFN sitt uttrykk. Figur 22 viser basis temaet til Keycloak.

EFN TEST REALM

Sign in to your account

Email

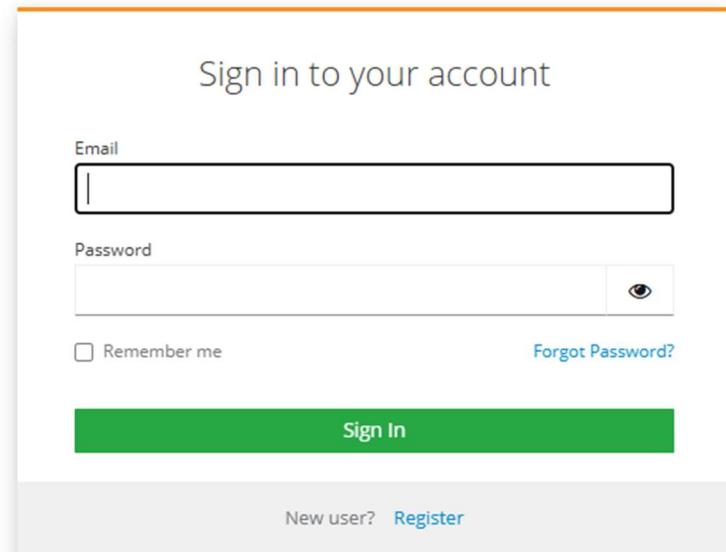
Password

 eye icon

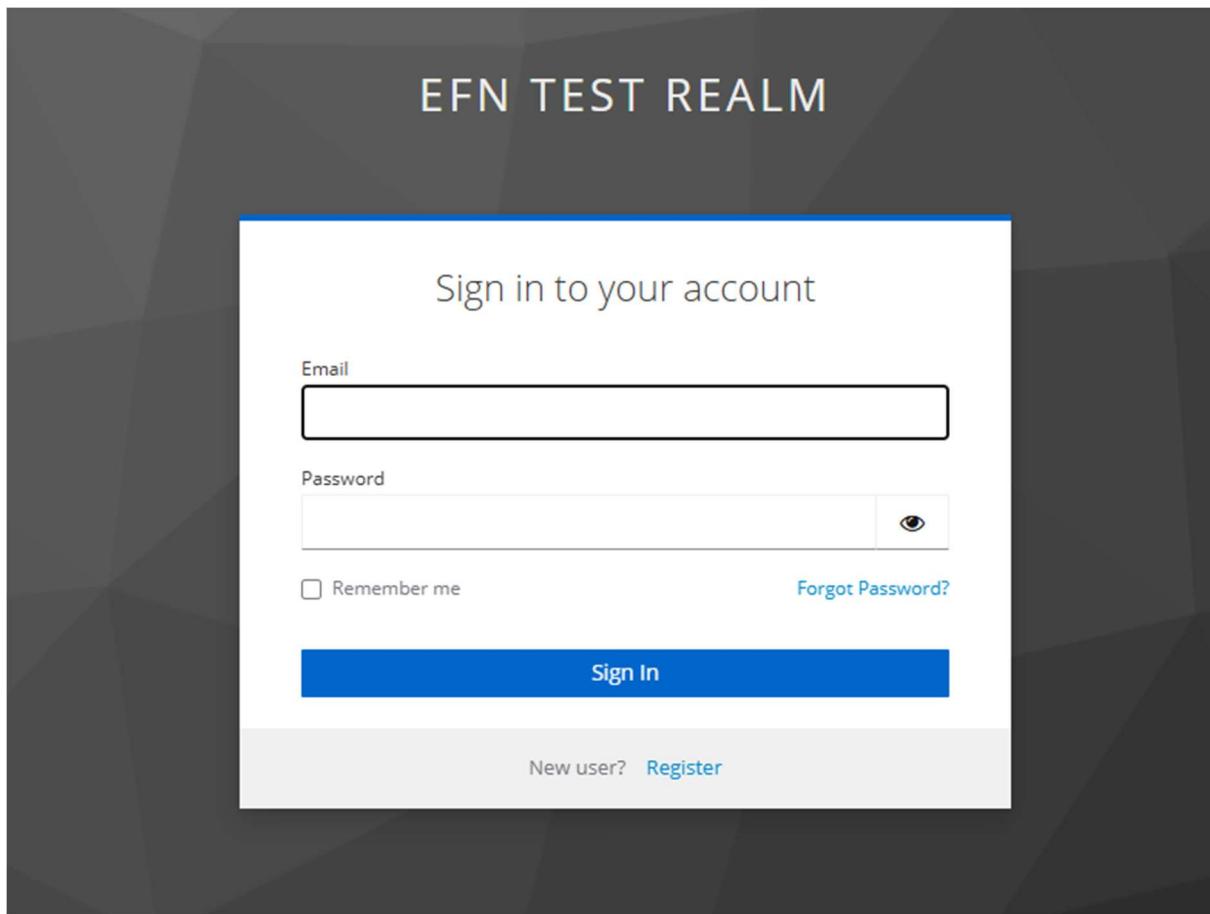
Remember me [Forgot Password?](#)

[Sign In](#)

New user? [Register](#)

The image shows a Keycloak login interface. At the top, the text "EFN TEST REALM" is displayed in orange. Below it, the heading "Sign in to your account" is centered. There are two input fields: one for "Email" and one for "Password", which includes a visibility toggle icon. A "Remember me" checkbox and a "Forgot Password?" link are located below the password field. A large green "Sign In" button is centered at the bottom of the form. At the very bottom, there is a note for new users with a "Register" link.

Figur 21 Keycloak brukerinnloggingsside



Figur 22 Keycloak brukerinnlogging side (med standard tema)

3.4.6.4 Brukerregistering

Brukerregisering håndteres også gjennom Keycloak, og med dette medføres enkel inputvalidering direkte i nettsiden. Alle feltene må være korrekt fylt ut før skjemaet kan sendes inn. Det vil automatisk hindre innsending dersom obligatoriske felter ikke er fylt ut, og vise en rød feilmelding med et faresymbol (se figur 24). For e-postfeltet utføres det i tillegg en syntaktisk sjekk som ser etter gyldig format, den sjekker om inputen inneholder «@», om det ikke gjør det blir det vist en egen feilmelding som sier at det er en ugyldig e-postadresse (se figur 25). Begge disse inputvalideringene er viktig for at «backend» ikke belastes med ufullstendige eller ugyldige forespørsler. Disse kunne ført til unødvendig trafikk og potensielle feilhåndteringssituasjoner.

Under registeringsfeltene har vi lagt inn en egendefinert tekst for å informere om behandling av personopplysninger (se figur 26). Her er «personvernerklæring» en klikkbar lenke som tar brukeren til en enkel HTML-side som viser personvernerklæring for medlemskap i EFN (se figur 27).

EFN TEST REALM

The screenshot shows a registration form titled "Register". It includes fields for First name, Last name, Email, Password, and Confirm password. Each field has an "eye" icon for password visibility. Below the fields is a note about data processing and a link to the privacy statement. At the bottom are "Back to Login" and "Register" buttons.

First name

Last name

Email

Password

Confirm password

Ved å opprette en konto og registrere deg på vår tjeneste, gir du ditt samtykke til at vi samler inn og behandler personopplysninger som beskrevet i denne [personvernerklæringen](#).

« Back to Login

Register

Figur 23 Brukerregistering

The screenshot shows the same registration form with validation errors. Each required field has a red border and a red exclamation mark icon. Error messages are displayed below each field: "Please specify this field." for First name, Last name, and Email, and "Please specify password." for Password.

First name

Please specify this field.

Last name

Please specify this field.

Email

Please specify email.

Password

Please specify password.

Confirm password

Figur 24 Viser input validering når bruker prøver å registrere uten å skrive noe i inputfeltene



Figur 25 Viser feilmelding om bruker skriver ugyldig e-postadresse

Ved å opprette en konto og registrere deg på vår tjeneste, gir du ditt samtykke til at vi samler inn og behandler personopplysninger som beskrevet i denne [personvernerklæringen](#).

Figur 26 Egendefinert tekst angående EFN sin personvernerklæring for medlemmer

Personvernerklæring for medlemskap i Elektronisk Forpost Norge

Gyldig siden: 14/03/2025
Oppdatert: 14/03/2025

Elektronisk Forpost Norge ("oss", "vi", eller "vår") opererer medlemskapssystemet <https://medlemskap.edn.no>, heretter referert til som "tjenesten". Denne siden beskriver vår policy angående innsamling, lagring og behandling av personlige data når du bruker vår tjeneste, samt valgene du har i forbindelse med disse datiene. Ved å bruke tjenesten samtykker du til innsamling og bruk av informasjon slik det er beskrevet i denne personvernerklæringen.

Innsamling og bruk av informasjon

Vi samler inn og behandler informasjon som er nødvendig for å tilby våre medlemskapstjenester, administrere tilgang, og sikre systemet.

Type av data som samles inn

Personlig data

- Full navn: For medlemsadministrasjon og personalisering av tjenestene.
- E-postadresse: For å sende viktig informasjon, påminnelser og nyheter. Brukes til loggning og identifikasjon.
- Passord: For å sikre tilgang til tjenesten. Brukes til loggning og behandles ved hjelp av sikkert hashing med salting for å beskytte mot ubesert tilgang.
- Medlemsnummer og medlemskapstype: For å administrere medlemskap og tilknyttede rettigheter.
- Betalingshistorikk og fakturaopplysninger: For å håndtere betalinger og abonnementadministrasjon.

Loggplat (IP adresse og påloggingshistorikk)

Vi samler automatisk inn loggdata via vår identitets- og tilgangsutstygsplattform (Keycloak), inkludert:

- Ditt IP-adresse ved loggning.
- Tidspunkt for pålogging og utlogging.
- Type enhet (f.eks. veldigkjet eller muslykket utlogging).

Samtykke

Ved å opprette en konto og registrere deg på vår tjeneste, gir du ditt samtykke til at vi samler inn og behandler personopplysninger som beskrevet i denne personvernerklæringen.

Bruk av data

Vi bruker innmønstrede data til følgende formål:

- Administrere medlemskap, påmeldinger og betalinger.

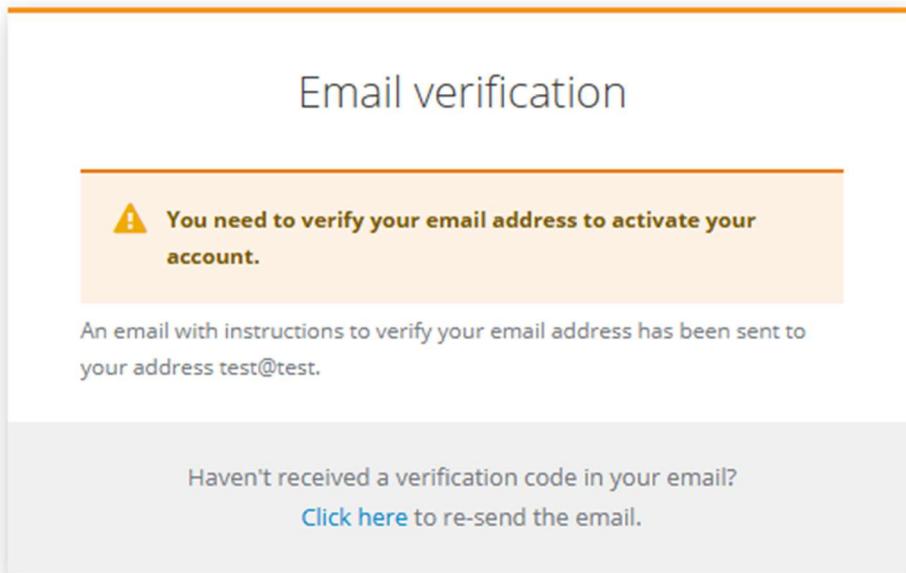
Figur 27 HTML-siden som innholder personvernerklæringen

3.4.6.5 Verifisering av bruker

Etter en vellykket registrering, krever systemet at brukeren verifiserer e-postadressen sin før kontoen aktiveres. Dette er en innstilling som man kan velge å aktivere i Keycloak-administrasjonsgrensesnitt, dette sørger for at ingen kontoer kan brukes før e-postadressen er bekreftet. Når brukeren fullfører registreringsskjemaet, blir de sendt videre til en side i Keycloak som informerer om at en bekrefteelse-post er sendt til oppgitt e-postadresse og at e-postadressen må bekreftes før kontoen blir aktiv (se figur 28).

Systemet var allerede satt opp slik at om man kjører det lokalt, benyttes Docker-bildet Haravich/fake-smtp-server, som gir mulighet for å simulere e-postflyt i testmiljøet. Verifiseringen vil derfor bli mottatt i «MailCatcher» istedenfor hos brukeren sin innboks (se figur 29). Når brukeren klikker på verifiseringens linken blir kontoen aktivert og man blir omdirigert til det digitale medlemskortet (se figur 20).

EFN TEST REALM



Figur 28 Informasjonsside om e-postverifisering

The screenshot shows an email in the Mailcatcher inbox. The header information is as follows:

From: <no-reply@efn.no>	To: <test@test>	Subject: Verify email	Received: Thursday, 15 May 2025 12:32:48 PM
-------------------------	-----------------	-----------------------	---

The email body content is:

```
Received: Thursday, 15 May 2025 12:32:48 PM
From: <no-reply@efn.no>
To: <test@test>
Subject: Verify email

Someone has created a EFN Test Realm account with this email address. If this was you, click the link below to verify your email address
Link to e-mail address verification
This link will expire within 5 minutes.
If you didn't create this account, just ignore this message.
```

Figur 29 Mailcatcher - en del av "Haravich/fake-smtp-server"

3.4.6.6 Glemt passord

Dersom en bruker har glemt passordet sitt, kan vedkommende bruke «Forgot password»-funksjonen fra innloggingsiden. Brukeren får da opp en melding som informerer om at en e-post er sendt (se figur 31). E-posten blir da sendt med en lenke der man kan tilbakestille passordet. (se figur 30).

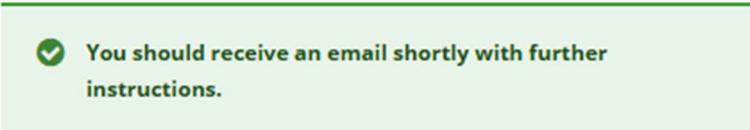
Someone just requested to change your EFN Test Realm account's credentials. If this was you, click on the link below to reset them.

[Link to reset credentials](#)

This link will expire within 5 minutes.

If you don't want to reset your credentials, just ignore this message and nothing will be changed.

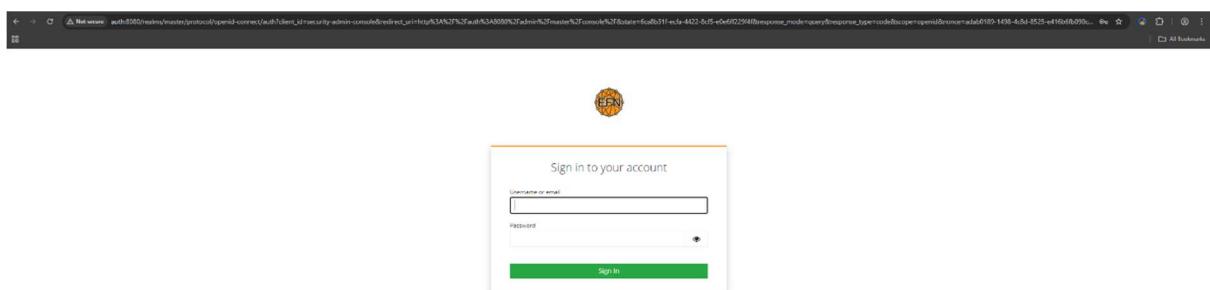
Figur 30 Viser e-post angående forandring av passord



Figur 31 Melding som blir vist etter å ha forespurt passord forandring

3.4.6.7 Administratorinnlogging

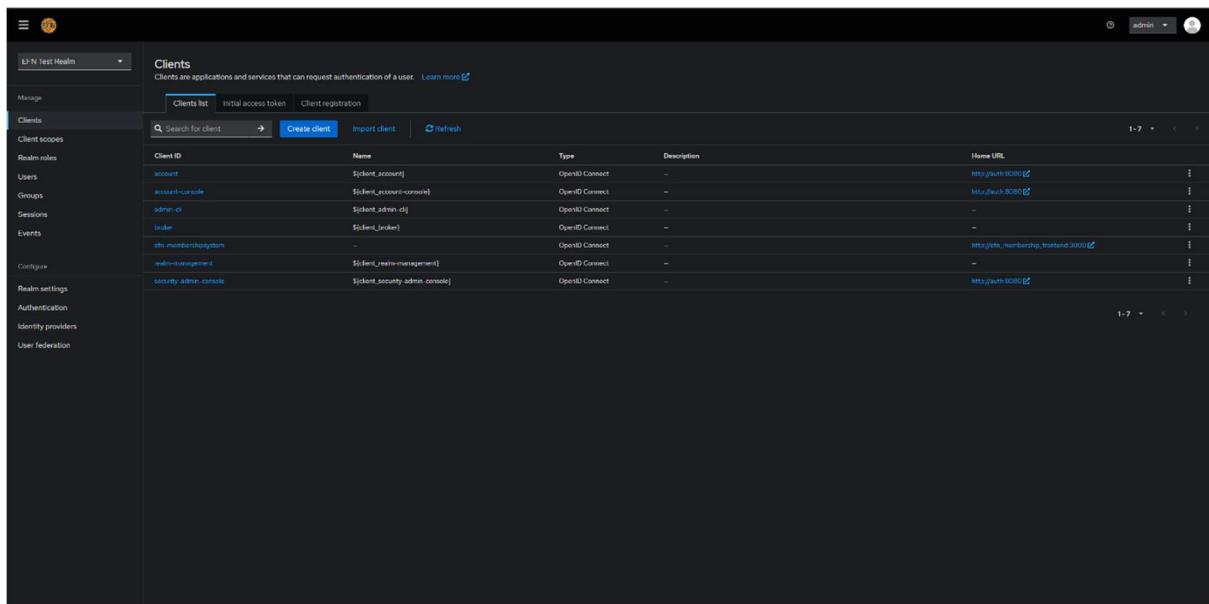
Administratorinnlogging er innloggingen for å få tilgang til Keycloak sitt administrative grensesnitt. Dette er en egen innloggingside som bare er ment for de som skal administrere Keycloak. Siden er stylet med samme fargevalg og visuelle tema som i brukergrensnittet, men her har vi lagt til EFN logo over innloggs vinduet (se figur 32). Når systemet kjøres lokalt er denne siden tilgjengelig på HTTP://auth:8080/, men i dev-, staging- og produksjonsmiljøene er denne siden ikke offentlig tilgjengelig. Når systemet kjøres opp i alt annet enn lokalt vil denne ligge bak «SSH-tunnel» med «SOCKS5-proxy». Dette sikrer at bare autoriserte brukere med tilgang til EFN serveren kan nå innloggsiden. Dette er gjort for å beskytte det administrative grensesnittet mot uautorisert tilgang.



Figur 32 Administratorinnlogging

3.4.6.8 Administrative grensesnitt (Keycloak)

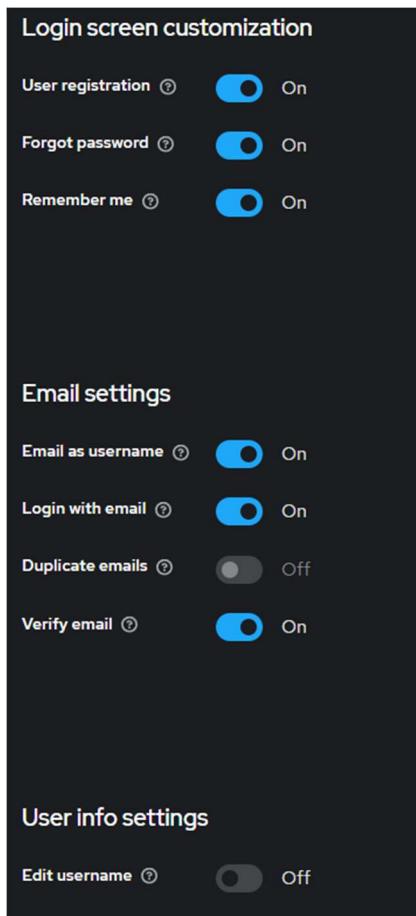
Keycloak sitt administrative grensesnitt gir full kontroll over autentiserings- og brukerstyringen i systemet (se figur 33). Her kan man blant annet opprette/slette «realms», forandre tema, forandre krav for bruker registering (se figur 34) og se/søke/filtrere brukere som er registrert. Vi har kun gjort minimale visuelle tilpasninger her, der Keycloak-logoen er byttet ut med EFN sin logo. I tillegg har vi gjort noen mindre justeringer på innstilingene på «realmen» får å løse «bugs» som oppstod under utviklingen. Bortsett fra dette er grensesnittet i stor grad uendret fra standardoppsettet.



The screenshot shows the Keycloak administrative interface for managing clients. The left sidebar has a dark theme with white text and icons. The main area has a light background. The title bar says "Clients" and "Clients are applications and services that can request authentication of a user". Below this is a search bar and a "Create client" button. The main table lists seven clients:

Client ID	Name	Type	Description	Home URL
account	Sjekk!_account	OpenID Connect	—	http://auth:8080/
account-console	Sjekk!_account-console	OpenID Connect	—	http://auth:8080/
admin-ds	Sjekk!_admin-ds	OpenID Connect	—	—
user	Sjekk!_user	OpenID Connect	—	—
user-membership	—	OpenID Connect	—	http://auth:membership.frontend:3000/
realm-management	Sjekk!_realm-management	OpenID Connect	—	—
security-admin-console	Sjekk!_security-admin-console	OpenID Connect	—	http://auth:8080/

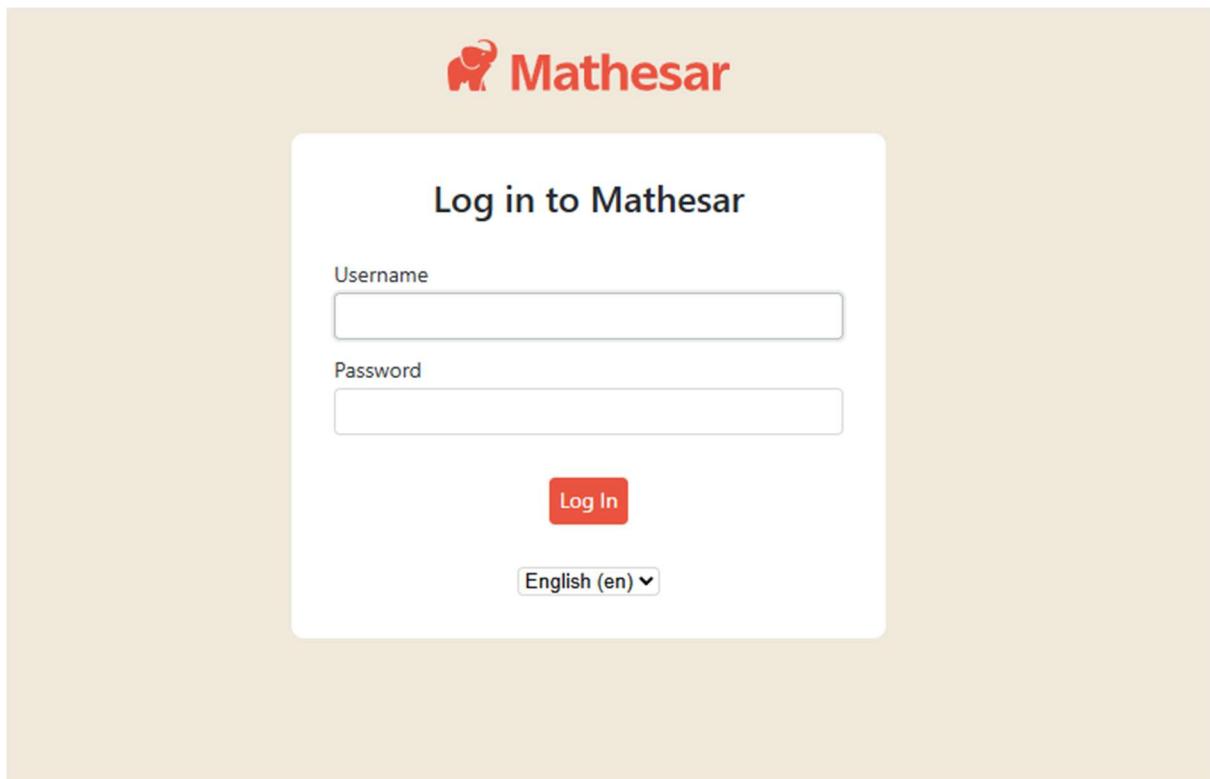
Figur 33 Administrative grensesnitt (Keycloak)



Figur 34 "realm" innstilinger bruker registering

3.4.6.9 Mathesar grensesnitt for innlogging

Innloggingsiden til Mathesar (se figur 35) er kun tilgjengelig for administratorer. Når systemet kjøres i dev-, staging- eller produksjonsmiljø er denne siden ikke offentlig tilgjengelig.



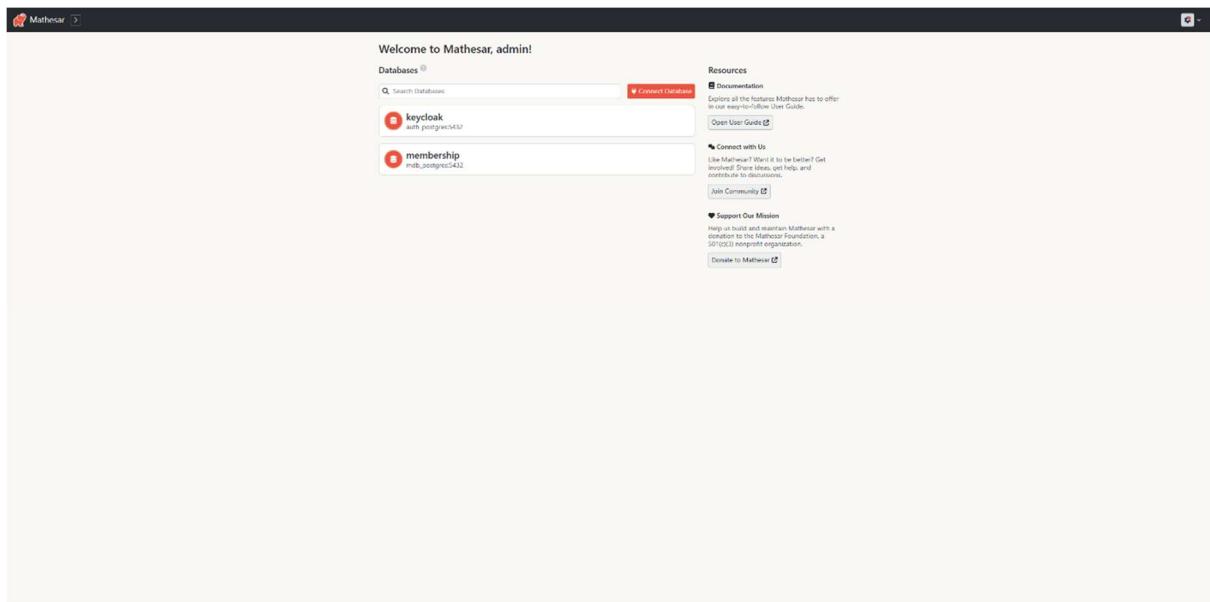
Figur 35 Grensesnitt for innlogging på Mathesar

3.4.6.10 Mathesar administrasjon av medlemsdata

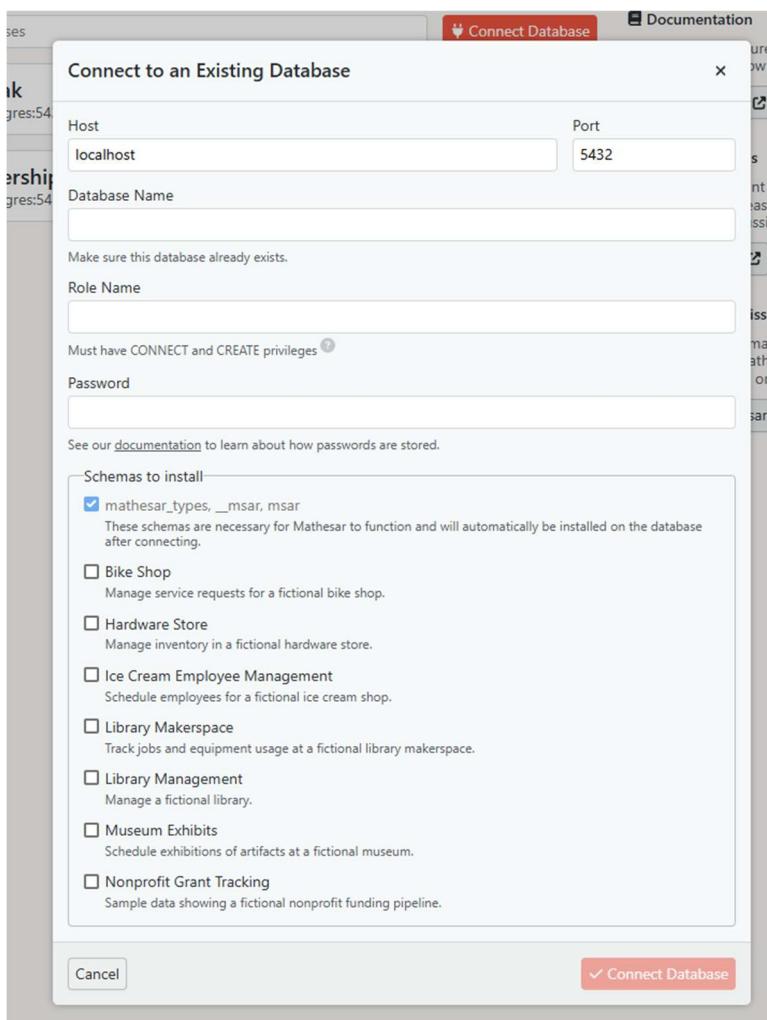
Etter innlogging i Mathesar møter brukeren et oversiktlig grensesnitt som viser tilgjengelige databaser (se figur 36), altså databaser som er manuelt koblet til via grensesnittet. Måten man kobler opp en ny database til Mathesar er ved å trykke på «Connect Database», så «Connect to an Existing Database» og fylle inn den nødvendige informasjonen inn i «pop-up» vinduet (se figur 37).

Vi har to databaser koblet opp til Mathesar: en for Keycloak, som inneholder systemets autentiseringsdata, og en egen «membership»-database som inneholder medlemsrelatert data for betalte medlemmer.

Mathesar gir et grafisk og brukervennlig grensesnitt for å håndtere data direkte i databasen, uten at man trenger å bruke SQL. Dette gjør det enkelt å blant annet legge til (se figur 38), endre (se figur 39) og filtrere data knyttet til medlemmer. Man kan også enkelt eksportere en hel tabell som en CSV-fil, og det er også mulig å gjøre dette med en aktiv filtrering (se figur 41)



Figur 36 Oversikt over tilgjengelige databaser



Figur 37 Input vindu for kobling med eksisterende database

	? id	≡ user_id	≡ year	≡ order_id	+ ↴
1					
2	179d74ec-ffe3-4eec ↗	member3	2025	35592f93bfcf4c5685f...	
3	555c3e82-4949-42cl ↗	member1	2025	35592f93bfcf4c5685f...	
4	93fa2a46-83b8-48d` ↗	member4	2025	35592f93bfcf4c5685f...	
5	d68ab9df-4f10-4b3f ↗	member2	2025	35592f93bfcf4c5685f...	
6	e2ca7029-2692-42d ↗	member5	2025	35592f93bfcf4c5685f...	

New records will be repositioned on refresh

6 *	DEFAULT ↗	NULL ↗	NULL ↴
+			

Figur 38 Viser hvordan man kan legge til nye data i databasen

6 *	00000000-0000-4000-8000-000000000000	MEMBER2	2025	35592f93bfcf4c5685f...
6	e2ca7029-2692-42d ↗	member5	2025	35592f93bfcf4c5685f...

New records will be repositioned on refresh

6 *	DEFAULT ↗	NULL ↗	NULL ↴
+			

Figur 39 Viser hvordan man kan endre data

	? id	Filter records	Sort	Group
1	e2ca7029-2692-42d ↗	where user_id contains member5		
+		+ Add New Filter ↴		

Figur 40 Viser hvordan man filtrerer data

Export the efnmembership table as a CSV file. Your current filters and sorting will be applied to the exported data.

Export Inspector

Table	Column	Record	Cell
-------	--------	--------	------

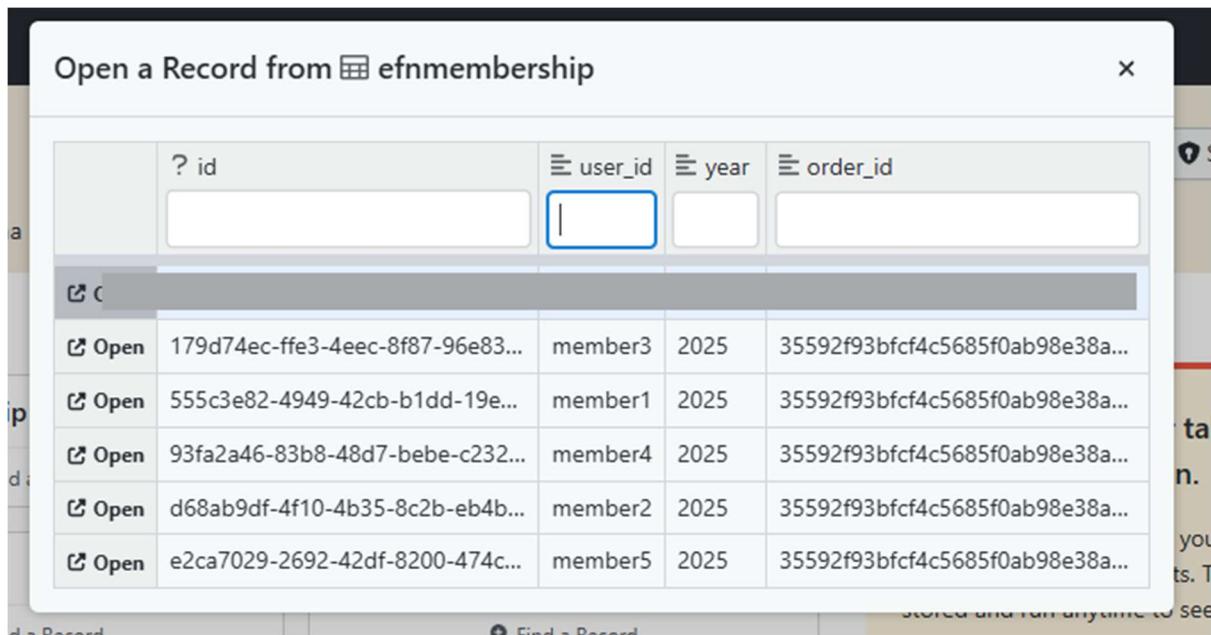
Figur 41 Viser at det er mulig å eksportere tabeller

3.4.6.11 Tabelloversikt i Mathesar

Når man klikker seg inn i for eksempel «membership»-databasen, får man oversikt over alle tabellene som inngår i denne databasen (se figur). Herfra kan man opprette nye tabeller etter behov, uten at det krever noen kunnskap om SQL. Når man trykker på «Find a Record» under en tabell så åpnes et vindu (se figur 43) som viser innholdet i tabellen. Her kan man fylle inn spesifikke verdier for å søke etter nøyaktig treff.

The screenshot shows the Mathesar database interface. At the top left is a red circular icon with a white 't' symbol. To its right, the word 'SCHEMA' is written in small capital letters, followed by 'public' in a larger, bold font. Below this, the text 'standard public schema' is displayed. In the center, the word 'Tables' is written in a dark green font. To the right of 'Tables' is a red button with a white plus sign and the text '+ New Table'. Below 'Tables', there are four table cards arranged in a 2x2 grid. Each card has a small icon of a table with three rows and three columns. The first card is labeled 'efnmembership', the second 'orders', the third 'product', and the fourth 'productinstance'. Underneath each table name is a blue 'Find a Record' button with a magnifying glass icon. To the right of each table name is a vertical ellipsis (three dots) icon.

Figur 42 Tabell visning

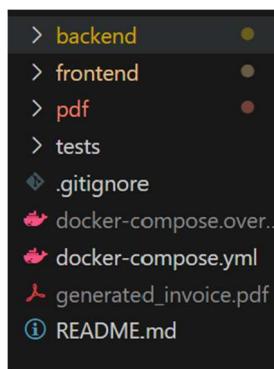


Figur 43 "Find in Record" vinduet

3.5 Betalingssystemets oppdeling

Dette kapitelet gir en grundig gjennomgang av hele betalingssystemet. Først forklares strukturen på programmet etterfulgt av en kort gjennomgang av hver enkelt del av programmet. Det er brukt kodeeksempler for å bedre vise hvordan koden fungerer, ikke alt er tatt med ettersom at dette ikke er nødvendig for å forklare hvordan programmet fungerer.

3.5.1 Mappestruktur og bruk av Docker



Figur 44 - Struktur betalingssystem

Betalingssystemet er et modulært system bestående av flere uavhengige komponenter som sammen gjør opp programmet. Hver komponent er gitt sin egen mappe og hver mappe sin egen Dockerfil. Når en av komponentene kjører i Docker kalles det en Docker container.

Containerne kan kjøres separat, både med og uten Docker, men de gir liten verdi hver for seg. I tillegg er det tidkrevende og upraktisk å starte dem én og én manuelt. Derfor brukes en Docker-compose fil som er en slags oppskrift som forteller hver enkelt container hvordan den skal settes opp og lar oss videre konfigurere hver container med miljøvariabler, volumer og fortelle containerene hvordan de skal samarbeide med hverandre og i hvilken rekkefølge de skal starte.

Nedenfor er det en figur som viser hvordan backend er konfigurerert i Docker-compose filen. “build: backend/” Containeren skal bygges fra Dockerfilen som ligger i mappen backend. Det er konfigureret mappe for volum altså et sted der data fra containeren lagres, slik at data ikke går tapt når containeren skrus av eller omstartes. Det er konfigurerert miljø-variabler som programmet trenger for å kjøre, men som ikke skal være leselig i koden. Det er konfigurerert at postgres_payment containeren skal være “healthy” før containeren kan kjøres og at containeren skal kjøres på port 8000. Det er konfigurerert liknende for hver komponent i programmet slik at alt kjøres opp på riktig hver gang instruksen “Docker compose up” er gitt.

```
▷ Run Service
backend:
  build: backend/
  volumes:
    - ./backend/src:/app/src
  environment:
    - DATABASE_URL=PLACEHOLDER
    - DOCKERIZED=true
  depends_on:
    - postgres_payment:
        condition: service_healthy
  networks:
    - backend_network
  ports:
    - "8000:8000"
```

Figur 45 - Docker-compose-fil

3.5.2 Frontend

Frontenden er en enkel react side der en med admin rettigheter manuelt kan gjøre endringer i databasesystemet. Nettsiden er fokusert på funksjonalitet og ikke design. Alle metoder i backend kan trigges fra frontend. I admin-siden er det en knapp “Refresh Invoices” som henter alle ubetalte fakturaer fra databasen og viser dem i en tabell for bruker. Denne funksjonen kjøres også når siden åpnes og hver gang en av de andre knappene er brukt til å

gjøre endringer i databasen. Knappen “Set to Paid” brukes til å manuelt endre betalingsstatus for faktura fra ubetalt til betalt. Knappen “Download PDF” laster ned en PDF-versjon av fakturaen basert på dataene gitt for fakturaen og bruker. Søkebaren lar administrator filtrere fakturaer etter ID. Knappen “Create new cash invoice” oppretter en ny faktura med fastsatte verdier. Denne funksjonen lar admin ta imot kontant betaling fra et medlem og opprette og betale en betaling for kunden. Dette lar betalingen bli loggført, og kunden får medlemskap på en riktig måte.

Invoice List

Invoice ID	Amount	Currency	Paid	Actions	
00017	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>
00018	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>
00019	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>
00020	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>
00021	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>
00022	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>
00023	100	NOK	No	<button>Set to Paid</button>	<button>Download PDF</button>

Figur 46 frontendside adminpanel

3.5.3 Backend

Backenden containeren vår kjører en Uvicorn-server som lytter på port 8000. Når det kommer inn en HTTP-forespørsel til porten videresendes forespørselen til fastAPI som styrer forespørselen til riktig fastAPI-endepunkt. Backend siden har endepunktene /fetch_unpaid og /webhooks/payment. Når et endepunkt mottar en forespørsel, vil forespørselen prosesseres og den etterspurte informasjonen blir sendt tilbake sett at alt er i orden.

For eksempel vil payment endepunktet ta imot en post forespørsel i JSON-format som inneholder et flagg "operation" og en hoveddel "body". Operasjons-flagget bestemmer hva som skal gjøres med "body". En JSON-forespørsel med et operasjonsflagg "markInvoicePaid" vil tilsi at "body" skal sendes til funksjonen "markInvoice". På denne

måten kan flere ulike funksjoner håndteres fra dette ene endepunktet. Med en oppskrift for hva som skjer med alle innkommende forespørsler er det enkelt å legge til nye funksjoner med minimalt av oppsett.

```
@app.post("/webhooks/payment")
async def handle_payment_webhook(
    db: Session = Depends(get_db),
    body: json = Depends(verify_webhook)
):
    operation = body["operation"]
    payload = body["body"]

    if operation == "cashPaymentInvoice":
        return(createInvoice(payload))
    elif operation == "markInvoicePaid":
        return(markInvoice(payload, db))
    elif operation == "fetchPdf":
        return(createPdf(payload, db))
    else:
        print("Bad request, invalid operation")
        raise HTTPException(status_code=400, detail="Bad request, invalid operation")
```

Figur 47 webhooks/payment endepunkt

Backend håndterer all kommunikasjonen med databasen gjennom sqlalchemy. Sqlalchemy er en Python bibliotek som lar deg gjøre databaseendringer i Python kode og beskytter ufullstendige oppdateringer. Når du gjør endringer i sqlalchemy vil endringene lagres i en sesjon og alt vil enten legges inn i databasen på en gang eller kan tilbakestilles ved behov. Dette hindrer ufullstendige databaseoppdateringer. Alle hovedfunksjonene i backend kobler seg til databasen og gjør enten endringer eller henter data. Nedenfor går jeg gjennom funksjonen markInvoice (se figur 48) og ser på hvordan den bruker informasjonen fra API-et og oppdaterer databasen på en trygg måte.

MarkInvoice-funksjonen tar inn en pakke og en instans av en åpen sqlalchemy-sesjon. Pakken består av verdiene betalings-id, faktura-id, mengde og valuta. Disse verdiene kommer i formen av en dict, en Python type for nøkkel-verdi-par. Funksjonen sjekker at alle nevnte verdier er en del av pakken, uten disse vil ikke funksjonen fungere og en feilmelding vil bli kalt. Funksjonen gjør en spørring til databasen og ser om det allerede finnes en betaling med id lik betalings-id. Hvis det allerede finnes en vil det bli returnert beskjed om at betalingen allerede har blitt prosessert. Hvis ikke vil det sjekkes at fakturaen eksisterer og at den har samme utestående beløp som det som er gitt i betalingen. Er alt i orden vil verdien “betalt” settes til “true” i faktura databasen og det blir satt et tidsstempel på tiden den ble betalt. En ny betaling vil så bli lagt inn i databasen. Med alt i orden vil “db.commit” bli kalt og alle databaseendringer blir lagt inn i databasen. Om det skjer en feil i løpet av funksjonen vil

“db.rollback” bli kalt og alle database-endringer vil bli slettet for å unngå å fylle databasen med feilaktig data.

```
def markInvoice(payment_data, db):
    try:
        # Validate required fields
        required_fields = ["payment_id", "invoice_id", "amount", "currency"]
        if not all(field in payment_data for field in required_fields):
            raise HTTPException(status_code=400, detail="Missing required fields")
        # Check for duplicate payment
        existing_payment = db.query(Payment).filter_by(payment_id=payment_data["payment_id"]).first()
        if existing_payment:
            return {"status": "already_processed"}
        # Get the invoice
        invoice = db.query(Invoice).filter_by(id=payment_data["invoice_id"]).first()
        if not invoice:
            raise HTTPException(status_code=404, detail="Invoice not found")

        # Validate amount matches
        if float(invoice.amount) != float(payment_data["amount"]):
            raise HTTPException(status_code=400, detail="Payment amount mismatch")

        # Validate currency matches
        if invoice.currency != payment_data["currency"]:
            raise HTTPException(status_code=400, detail="Currency mismatch")

        # Update invoice status
        invoice.paid = True
        invoice.paid_at = datetime.now()

        # Create payment record
        new_payment = Payment(
            payment_id=payment_data["payment_id"],
            invoice_id=payment_data["invoice_id"],
            amount=payment_data["amount"],
            currency=payment_data["currency"],
            payment_method = payment_data["payment_method"]
        )
        db.add(new_payment)
        # Commit changes
        db.commit()
        return {"status": "success", "invoice_id": payment_data["invoice_id"]}

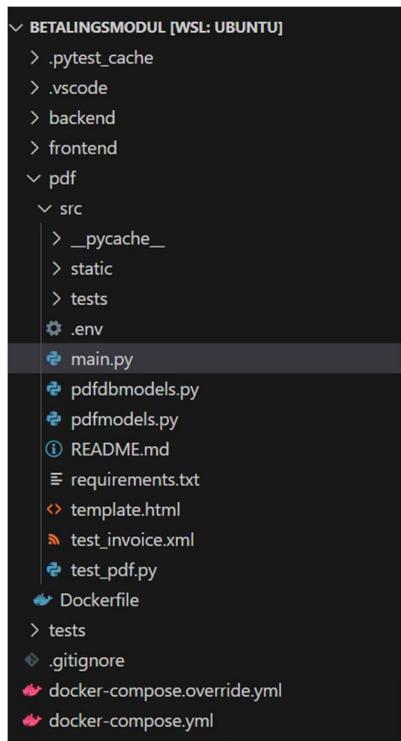
    except json.JSONDecodeError:
        raise HTTPException(status_code=400, detail="Invalid JSON payload")

    except HTTPException as http_err: # Catch HTTPException specifically
        raise http_err # Let it propagate as is (e.g., 400 or 404)

    except Exception as e:
        db.rollback()
        print(e)
        raise HTTPException(status_code=500, detail=str(e))
```

Figur 48 “markInvoice”

3.5.4 PDF-modul



Figur 49 oppsett PDF-modul

PDF-modulen kjører også en Uvicorn server med FastAPI. Den tar inn en post forespørsel og leverer tilbake en PDF. De mest elementære funksjonene i PDF-programmet er «parse_xml(...)» denne bruker ElementTree-biblioteket for å hente ut data fra XML -filer med UBL format. Generate_PDF(...) bruker et bibliotek kalt Jinja2 til å fylle ut en HTML mal som så blir rendret som PDF format et bibliotek som heter WeasyPrint. Dette returnerer så en «StreamingResponse» som er et ferdig PDF-dokument skrevet ut ifra input dataen.

I Figur 49 ser man to filer, en som heter «template.html» og en som heter test_invoice.xml. template.html inneholder malen som brukes til å bestemme formatet til PDF-er. test_invoice.xml ble brukt i utviklingsfasen og til tester og er et eksempel på en godkjent XML input i UBL format.

Metoden «get_invoice_settings(...)» blir brukt for rate-limiting og API-token. Denne metoden justerer fakturaen basert på API-tokenet. API-tokenet er primærnøkkelen til en tabell i databasen og vil hente ut verdiene: fakturainnstillinger, selskapsinfo, logo og standardvaluta. Verdiene tilhørende API-nøkkelen blir brukt til å fylle inn organisasjonsinformasjon i PDF-en. Idéen er at forskjellige organisasjoner skal kunne legge

inn sin data og at denne kan brukes flere ganger. En faktura som er knyttet til en organisasjon med en API-token vil omgå rate-limiteren til API-endepunktet «POST /generate_invoice».

Denne er egentlig satt til å godta maks 5 kall per minutt. Rate-limiteren er satt for å ikke overbelaste serverne og for å unngå DDOS og sabotasjeforsøk.

I figur 50 ser du en PDF generert av PDF-modulen med API-tokenet til EFN.



Figur 50 Generert PDF fra PDF-modul

3.5.5 Postgresql db

Databasen bygges ifra strukturen i dbmodels.py som ligger i backend mappen. Databasen bygges med 3 tabeller. Invoice, Payment og InvoiceSetting. Alle opprettes som klasser med sqlalchemy sin "Base"(figur 51), satt for å deklarere at de skal gjøres om til sql-tabeller. Invoice inneholder informasjon om beløpet som skyldes og en boolean for å si om beløpet er betalt eller ikke.

Payment inneholder informasjon om alle betalinger. Hver betalt faktura vil ha en tilhørende betaling. Alle betalinger lagres, og det blir en slags logg for systemet. InvoiceSettings er brukt av PDF systemet. Den brukes til å lagre data om organisasjoner. Navn på organisasjon, adresse, logo og annen organisasjonsinformasjon som kreves i en PDF.

```

class Invoice(Base):
    __tablename__ = "invoices"
    id = Column(String, primary_key=True)
    amount = Column(Numeric(10, 2))
    currency = Column(String(3))
    paid = Column(Boolean, default=False)
    paid_at = Column(DateTime(timezone=True))

class Payment(Base):
    __tablename__ = "payments"
    id = Column(Integer, primary_key=True)
    payment_id = Column(String(50), unique=True)
    invoice_id = Column(String, ForeignKey("invoices.id"))
    amount = Column(Numeric(10, 2))
    currency = Column(String(3))
    created_at = Column(DateTime, default=datetime.now())
    payment_method = Column(String(50))

class InvoiceSettings(Base):
    __tablename__ = "invoice_settings"
    id = Column(Integer, primary_key=True)
    api_token = Column(String(255), unique=True, nullable=False)
    currency = Column(String(10))
    logo_url = Column(String(255))
    custom_message = Column(String)
    supplier_name = Column(String(255))
    supplier_address = Column(String)
    supplier_contact = Column(String(255))
    supplier_tax_id = Column(String(100))
    created_at = Column(DateTime(timezone=True), default=lambda: datetime.now(timezone.utc))
    updated_at = Column(DateTime(timezone=True), default=lambda: datetime.now(timezone.utc)),

```

Figur 51 database betalingssystem

Databasen inneholder ingen data om medlemmer. Det er tenkt at medlemskapssystemet skal holde styr på hvilke betalinger og fakturaer som er knyttet til hvilke medlemmer.

3.5.6 Sikkerhet i backend

Docker containere kan ikke kommunisere direkte med hverandre. For å få til dette brukes for det meste API-er. Koblingen mellom frontend og backend vil etter hvert gå over fra å være en kobling fra container til container i samme system til å være koblet sammen gjennom to separate systemer over nett. Denne koblingen må være sikker, derfor er det valgt å bruke HMAC autentisering for å forsikre trygg ende-til-ende pakke-levering.

I figur 52 vises det hvordan en HMAC signatur generes i frontend og brukes i API-forespørselen. Funksjonen generateSignature bruker en hemmelig nøkkel til å lage en HMAC signatur basert på pakken som sendes. Signaturen vil være lik om samme nøkkel og innhold er brukt til å lage signaturen. Pakken sendes så til API-et med signaturen satt i headeren.

```

const generateSignature = (payload) => {
  const secret = WEBHOOK_SECRET;
  const sortedPayload = JSON.stringify(payload, Object.keys(payload).sort());
  return CryptoJS.HmacSHA256(sortedPayload, secret).toString(CryptoJS.enc.Hex);
};

export const fetchUnpaidInvoices = async () => {
  try {
    const payload = {
      timestamp: new Date().toISOString()
    };

    const signature = generateSignature(payload);

    const headers = {
      'X-Signature': signature,
      'Content-Type': 'application/json',
      'Operation': 'fetchUnpaidInvoices'
    }

    const response = await api.post('/fetch_unpaid', payload, {headers} );
    const data = JSON.parse(response.data.data);
    return data;
  } catch (error) {
    console.error('Error fetching invoices:', error);
    throw error;
  }
};

```

Figur 52 generateSignature

Når pakken mottas i et av endepunktene vil den først bli sendt gjennom en funksjon som verifiserer den sendte signaturen (funksjon verify_webhook, visst i figur 53). Funksjonen tar inn hele HTTP forespørselen og sjekker at innkommende forespørsel er ekte og uendret. For å få til dette bruker den samme hemmelige nøkkelen brukt i frontend. Forespørselen sin kropp oversettes fra bytes til streng og strengen blir videre omgjort til JSON format. Ut ifra hemmelig nøkkelen og JSON-kroppen genereres det en HMAC signatur. Hvis ingen endringer er gjort i pakken vil signaturen være helt lik signaturen fra frontend. Når signaturen er validert sendes kroppen videre til bruk for andre funksjoner.

```

async def verify_webhook(request: Request):
    required_fields = ["X-Signature", "Operation"]
    if not all(field in request.headers for field in required_fields):
        raise HTTPException(status_code=400, detail="Missing required fields")

    signature = request.headers.get("X-Signature")
    operation = request.headers.get("Operation")
    body = await request.body()
    str_body = body.decode()

    #print(f"body {body}")
    try:
        parsed_body = json.loads(str_body)
    except json.JSONDecodeError:
        raise HTTPException(status_code=400, detail="Invalid JSON body")

    reconstructed_body = json.dumps(parsed_body, separators=', ', sort_keys=True)
    #print(f"parsed_body : {parsed_body}, reconstructed_body : {reconstructed_body}")
    secret = os.getenv("WEBHOOK_SECRET").encode()
    expected_sig = hmac.new(secret, reconstructed_body.encode(), hashlib.sha256).hexdigest()
    #print(f"Expected_sig: {expected_sig}, actual_sig: {signature}")
    if not hmac.compare_digest(signature, expected_sig):
        print("Bad signature")
        raise HTTPException(status_code=400, detail="Invalid signature")

    #print("verify er good")
    return {"body" : parsed_body, "operation" : operation}

```

Figur 53 verify_webhook

4. Testdokumentasjon

4.1 Introduksjon

Dette dokumentet beskriver testingen vi har gjennomført for betalingssystemet og medlemskapssystemet. Her finner du informasjon om hvorfor vi har testet, hva vi har testet og hvordan vi har testet det.

4.2 Hvorfor testing er viktig for dette prosjektet

Testing har vært helt nødvendig i dette prosjektet siden vi jobber med et betalingssystem der feil kan få alvorlige konsekvenser for brukernes økonomi.

Grundig testing hjelper oss å:

- Unngå økonomiske tap for brukerne
- Sikre at systemet er pålitelig og sikkert
- Oppfylle lovkrav for betalingssystemer
- Se hva programmet tåler å ikke tåler
- Simulere situasjoner som kan oppstå i produksjon, men som ikke oppstår naturlig i staging
- Se om nye feil oppstår i gammel kode ved innføring av ny funksjonalitet

Ved å teste kontinuerlig kan vi oppdage og fikse problemer før de blir alvorlige. Testing gjør det også lettere å se hvordan systemet oppfører seg i ulike situasjoner og forbedre feilhåndteringen.

4.3 Tester

4.3.1 Tester I Medlemskapssystemet

4.3.1.1 Integrasjontest av backup og gjenoppretting av live-database

Det overordnede målet med denne testingen var å kvalitetssikre den opprinnelige backup- og restore-funksjonaliteten i systemet. Dette var en backup løsning som EFN hadde utviklet for å ta regelmessige sikkerhetskopier av databasen over betalte medlemmer. Det er svært kritisk at vi testet backup-funksjonaliteten for å sikre at ingen data går tapt ved systemfeil. Dersom testene avdekket feil eller svakheter, måtte vi utvikle og forbedre løsningene videre for å sikre en mer pålitelig datagjenoppretting.

Hele backup-prosessen ble ikke testet, kun resultatet av en gjenopprettning. Testen ble utført manuelt ved hjelp av et Python-skript som benytter unittest, og hadde som mål å sjekke om databasetabeller var på plass og at bestemte forventede data faktisk var riktig etter restore-operasjonen. Forventede data er testdata vi manuelt la inn på databasen i staging miljøet. Vi gjenopprettet databasen i det lokale miljøet ved hjelp av både manuelle og automatisk backup-filer som var tatt fra staging miljøet.

Testen verifiserer konkret at databasetabellene «efnmembership», «orders», «product» og «productinstance» eksisterer etter gjenopprettningen. I tillegg kontrolleres det at tabellen «efnmembership» inneholder nøyaktig seks brukere, og at en spesifikk bruker med en gitt ID finnes blant disse. Til slutt sjekkes det at alle rader i «orders» har en gyldig «order_id», det vil si at ingen rader har NULL-verdier satt som «order_id».

4.3.1.2 Integrasjontest av e-postutsending ved nytt betalt medlem

Denne testen ble brukt for å teste e-postutsendingen via «send_email.py», både under og etter utviklingen. Testen gjennomføres ved hjelp av Python-skriptet «test_email_sending.py» (se figur 54), som manuelt oppretter et nytt betalt medlem i databasen ved å kalle `insert_membership()`-funksjonen. Når denne funksjonen kjøres triggется automatisk et kall til `send_templated_email()` fra «send_email.py», som forsøker å sende en e-post til adressen som er oppgitt som argument når testskriptet kjøres. Testen ble ansett som vellykket når e-posten ble sendt uten feil og mottatt korrekt, og var veldig verdifull under utviklingen av «send_email.py».

```

❸ test_email_sending.py ✘
home > aho > membershipsystem-main > devserver > decriptors > ❸ test_email_sending.py
1 #!/usr/bin/env python3
2 """
3 Test script for testing email sending when creating an EFN membership.
4
5 Where and how this script is used:
6 - This script is run manually during development or testing to verify that email sending is working as expected.
7 - It calls 'insert_membership()' from postgres_interface.py to insert a new membership into the database.
8 - The 'insert_membership()' function itself handles sending the confirmation email
9 | by calling 'send_templated_email()' from send_email.py internally after inserting the membership.
10 - The database connection arguments are loaded from .env files (e.g., dev-localhost.env, staging.env, etc.).
11
12 What it tests:
13 - That inserting a new EFN membership (via `PostgresPersistence.insert_membership()`)
14 | correctly inserts a database record and sends a "membership_confirmation" email
15 | to the specified email address.
16 - The membership remains in the database after the test (no cleanup is performed).
17
18 Environment Variables:
19 - MBD_POSTGRES_DB_NAME: Name of the PostgreSQL database (default: "membership").
20 - MBD_POSTGRES_DB_USER: Database user name (default: "membership").
21 - MBD_POSTGRES_DB_PASSWORD: Database user password (default: "password").
22 - DB_HOST: Hostname of the PostgreSQL server (default: "mdb_postgres").
23 - DB_PORT: Port for the PostgreSQL server (default: "5432").
24
25 Usage:
26 | python3 test_email_sending.py [email_address]
27 Example:
28 | python3 test_email_sending.py test@example.com
29 """
30
31 import os
32 import sys
33 import uuid
34
35 # The following library imports are used for:
36 # - os: used to fetch environment variables to configure the database connection.
37 # - sys: used to get command-line arguments (email address) passed to the script.
38 # - uuid: used to generate a unique order ID
39
40 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "../src")))
41
42 # Import necessary modules
43 from persistence.postgres_interface import PostgresPersistence
44 from purchase_processing.persistence.persisted_domain_model import EfnMembership
45
46 def main():
47
48     # Check if an argument was provided
49     if len(sys.argv) != 2:
50         print("ERROR: Exactly one argument is required.")
51         print(__doc__)
52         return
53
54     arg = sys.argv[1]
55
56     # Handle help option
57     if arg in ["-h", "--help"]:
58         print(__doc__)
59         return
60
61     # Validate email format (basic check for '@')
62     if "@" not in arg:
63         print("ERROR: Invalid email address '{arg}'. Must contain '@'.")
64         print("Usage: python test_email_sending.py <email_address>")
65         return
66
67     test_email = arg
68
69     # Use the database configuration provided
70     db_name = os.environ.get("MBD_POSTGRES_DB_NAME")
71     db_user = os.environ.get("MBD_POSTGRES_DB_USER")
72     db_host = os.environ.get("DB_HOST", "mdb_postgres")
73     db_password = os.environ.get("MBD_POSTGRES_DB_PASSWORD")
74     db_port = int(os.environ.get("DB_PORT", "5432"))
75
76
77     # Create database connection
78     try:
79         db = PostgresPersistence(
80             database=db_name,
81             user=db_user,
82             host=db_host,
83             password=db_password,
84             port=db_port
85         )
86         print("Database connection successful")
87     except Exception as e:
88         print(f"Error connecting to database: {e}")
89         return
90
91     try:
92
93         # Generate a unique order ID
94         test_order_id = f"test_order_{uuid.uuid4().hex[:8]}"
95         test_year = "2025"
96
97         # Create a test membership
98         membership = EfnMembership(
99             user_id=test_email, # Using email as user_id
100            year=test_year,
101            order_id=test_order_id
102        )
103
104         # Call insert_membership which should trigger the email
105         db.insert_membership(membership)
106
107     except Exception as e:
108         print(f"Error during test: {e}")
109     finally:
110
111         # Remember it the membership in the database stays there after test
112         print("Test completed")
113         db.close()
114
115     if __name__ == "__main__":
116         main()

```

Figur 54 “test_email_sending.py”

4.3.2 Tester I Betalingssystemet

4.3.2.1 Enhetstester betalingssystemet

Mye logikk og metoder er lagt inn i PDF-modulen. Enhetstestene tester de forskjellige enhetene hver for seg. Med unittester testes det på lavest mulig nivå (*Guide to Agile Practices*, 2012, Python Software Foundation, u.å.-a). Alle testene ligger i testmappen, denne er utenfor Docker miljøet fordi testene ikke har noe med den faktiske kjøringen av programmet å gjøre. Under tests mappen har vi en fil test_PDF.py for enhetstesting i PDF-modulen. Denne filen består av flere testklasser, det er en klasse per funksjon i programmet, hver klasse inneholder flere tester for denne funksjonen. Enhetstestene tester at de forskjellige funksjonene feiler når de skal og fungerer når riktig data er gitt som input.

Python sitt unittest bibliotek er brukt for å gjennomføre testene. En enhetstest er typisk delt i tre deler: oppsett, gjennomføring og resultat. I figur 55 vises en test for å sjekke at verify_webhooks har mottatt en forespørsel med alle nødvendige felter fyllt ut. Først settes det

opp hvilke data som skal brukes for å teste funksjonen, det lages en falsk forespørsel som inneholder all den riktige informasjonen, men mangler feltet «operation». Den falske informasjonen settes så inn i en falsk forespørsel som er formatet verify_webhooks tar.

```
async def test_verify_webhook_missing_fields(self):  
    payload = datetime.now().isoformat() + "Z"  
    fake_body = json.dumps({"timestamp": payload}, separators=(',', ':'), sort_keys=True)  
  
    mock_request = AsyncMock(spec=Request)  
    mock_request.headers = {}  
    mock_request.body.return_value = fake_body.encode()  
  
    with self.assertRaises(HTTPException) as cm:  
        await verify_webhook(mock_request)  
  
    self.assertEqual(cm.exception.status_code, 400)  
    self.assertEqual(cm.exception.detail, "Missing required fields")
```

Figur 55 – test_verify_webhook

Testen vil så gjennomføre selve funksjonen, den tar inn svaret og sjekke at det stemmer med forventet feilmelding. I dette tilfellet en feilmelding med status kode 400 og beskjed: nødvendige felt mangler.

I PDF blir alle 12 tester gjennomført slik som de skulle som man kan se i figur 56.

```

pdf > src > test_pdf.py > TestParseXml
32
33 class TestGeneratePDF(unittest.TestCase):
34
35     def setUp(self):
36         self.invoice = Invoice(
37             id="123",
38             issueDate="2025-04-01",
39             dueDate="2025-04-15",
40             supplier="Supplier Inc.",
41             customer="Customer LLC",
42             customerAddress="123 Street",
43             customerCity="Oslo",
44             customerCountry="Norway",
45             invoiceLine=[
46                 {'itemDescription': "Test Item",
47                  'quantity': {'value': 1, 'unitCode': 'EA'},
48                  'price': {'value': 100, 'currencyID': 'USD'},
49                  'lineExtensionAmount': {'value': 100, 'currencyID': 'USD'}
49
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS bash -
test_get_invoice_settings_success (__main__.TestGetInvoiceSettings.test_get_invoice_set
test_invalid_xml (__main__.TestParseXml.test_invalid_xml) ... [2025-05-19 12:43:09] [ER
o element found: line 1, column 9
ok
test_missing_fields (__main__.TestParseXml.test_missing_fields) ... ok
-----
Ran 12 tests in 0.047s
OK

```

Figur 56 Resultat av enhetstest for pdf

I prosjektet har vi hatt som mål at all koden vi skriver skal være dekket av enhetstester.

Senere i prosjektets løp har vi ikke hatt lov til å merge til main før vi kan dokumentere at enhetstestene fungerer for de nye kodeendringene.

4.3.2.2 Stresstest av PDF-APIet

I tillegg til enhetstestene har vi også kjørt en stresstest av PDF-APIet. Det som er viktig er å teste om systemet vårt tåler den tiltenkte bruken, og at det kan håndtere et stort antall forespørsler. For å teste dette brukte vi et verktøy som heter Apache Benchmark (Apache HTTP Server Version 2.4. (u.å.)). Vi kjørte opp Docker-miljøet og sendte 1000 forespørsler til PDF-endepunktet, 50 om gangen. Stresstesten ble kjørt på to forskjellige maskiner, en med 16 GB ram og en med 32 GB ram. Resultatene var særdeles bedre på maskinen med 32 GB ram. Et utdrag fra den testen som ble kjørt på 32 GB ram maskinen kan du se i figur 57. På 32 GB maskinen feilet omtrentlig 1/10 av testene, imens på 16 GB maskinen feilet mer enn ½ av testene. Dette viser at PDF-programmet vårt er veldig ressurskrevende og ikke tåler mange brukere samtidig.

En annen svakhet som vi avdekket ved stresstesten, var det faktum at vi har problemer med rate-limiteren. Stresstesten gjennomførtes etter at vi hadde konkludert kodingen. Rate-limiteren skulle ha lagt til en grense på 5 forespørsler per IP-adresse per minutt. Så her har vi forøvrig avdekket at den ikke fungerer. Dette er noe vi har notert oss og videreført til

oppdragsgiver. Dette hadde vi ikke avdekket hvis vi ikke hadde kjørt stresstest, og viser viktigheten av testing. Vi hadde trolig avdekket dette ved en senere anledning i prosjektet om vi ikke hadde stoppet da vi gjorde.

```
Benchmarking localhost (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests

Server Software:      uicorn
Server Hostname:     localhost
Server Port:          8001

Document Path:        /generate_invoice/
Document Length:     534 bytes

Concurrency Level:    50
Time taken for tests: 21.986 seconds
Complete requests:   1000
Failed requests:      110
    (Connect: 0, Receive: 0, Length: 110, Exceptions: 0)
Non-2xx responses:   1000
Total transferred:   684879 bytes
Total body sent:     2371000
HTML transferred:   533879 bytes
Requests per second: 45.48 [#/sec] (mean)
Time per request:    1099.322 [ms] (mean)
Time per request:    21.986 [ms] (mean, across all concurrent requests)
Transfer rate:       30.42 [Kbytes/sec] received
                      105.31 kb/s sent
                      135.73 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1  1.1     1      6
Processing:    20  1080 141.2   1095   1350
Waiting:       18   693 286.3    709   1348
Total:         26  1081 140.7   1098   1350

Percentage of the requests served within a certain time (ms)
  50%  1098
  66%  1126
  75%  1145
  80%  1164
  90%  1225
  95%  1266
  98%  1271
  99%  1271
100%  1350 (longest request)
```

Figur 57 - Resultat av stresstest

4.4 Testevaluering

Om vi skulle konkludere og evaluere test arbeidet så kan vi konkludere at vi skulle ha startet testing på enda tidligere enn det vi gjorde. Vi burde ha hatt flere kodegjennomganger med oppdragsgiver og tidligere satt høye krav for testing av systemet.

5. Avsluttende del

5.1 Diskusjon om faglige utfordringer og valg

5.1.1 Valg I Medlemskapssystemet

Valg av medlemshåndtering

EFN ønsket seg et verktøy for å kunne vise innholdet i medlemsdatabasen. Ønsket var å få et brukervennlig behandlingsverktøy, som man kunne se, endre og slette data fra databasen uten å ha kjennskap til SQL. Oppdragsgiver hadde allerede sett på mulige åpen kildekode løsninger, og foreslo Mathesar som et aktuelt alternativ basert på deres vurdering. Vi begynte derfor å lese oss opp på Mathesar og undersøkte alternative løsninger som kunne vært bedre egnet for medlemskapssystemet. Vi ble stående mellom to alternativer, Mathesar eller Baserow. Begge er åpen kildekode og kan kjøres fra egen maskin, men de har forskjellige styrker og svakheter.

Baserow tilbyr en fleksibel og kraftig plattform med støtte for flere brukstilfeller, og er bedre egnet for brukere som ønsker mer kontroll, tilpasning og avanserte funksjoner. Det tilbyr avanserte funksjoner som API-er, tilpassede skjemaer og dra-og-slipp-grensesnitt.

Mathesar derimot mer laget med fokus på brukervennlighet, med en enklere tilnærming til databaseadministrasjon. Det retter seg mot brukere som ønsker å jobbe med databaser uten å måtte forholde seg til kompleks SQL eller tradisjonelle databasesystemer.

Mathesar sitt fokus på PostgreSQL-administrasjon uten unødvendig funksjoner gjorde det mer målrettet for vårt bruk. Baserow med sin brede funksjonalitet ville introdusert kompleksitet vi ikke trengte. Viktigste av alt var å se på ressursbruk mellom de to, vi vil prøve å holde dette så lavt som mulig for å gi systemet vårt mer fleksibilitet i hvem som kan benytte det. Ser vi på figur 58 og figur 59, kan vi se at Baserow bruker over ti ganger mer RAM og over tusen ganger mer CPU når den kjøreres lokalt.

Dette gjorde Mathesar til det mer passende valget for oss, ikke bare på grunn av sin enkle og målrettede design, men også fordi den gir oss betydelig bedre ytelse og lavere ressursbruk. Dette gjør det mulig å kjøre systemet på enklere maskinvare, noe som igjen øker tilgjengeligheten for flere brukere.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
71d8ac76ef72	mathesar	0.09%	114.5MiB / 15.58GiB	0.72%	1.7kB / 126B	0B / 0B	9

Figur 58 Ressursbruk av Mathesar

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
c837670ac59f	baserow	98.36%	1.262GiB / 15.58GiB	8.10%	1.17kB / 126B	0B / 0B	58

Figur 59 Ressursbruk av Baserow

Valg rundt skinning av keycloak

Et av valgene vi måtte ta underveis i prosjektet var hvordan vi skulle tilpasse utseendet på Keycloak så det skulle ligne mer på medlemskapssiden vi hadde i React, særlig når det gjelder farger og uttrykk. Keycloak kommer med et ferdig design som funker greit, men skiller seg ganske mye fra stilene vi brukte på medlemssiden.

Vi så på flere måter å løse dette på. En mulighet var å lage et helt eget tema fra bunnen av med FreeMarker-maler og egne CSS- og JavaScript-filer. Det gir mye frihet, men tar også mye tid og er mer å vedlikeholde. Vi vurderte også Keycloakify, som lar deg bygge temaer med React og TypeScript. Det er et kraftig verktøy og gir god kontroll, men for vårt bruk ble det rett og slett for omfattende.

Til slutt gikk vi for en enkel løsning hvor vi bare brukte CSS for å overstyre kun de delene av designet vi ønsket å endre, mens resten av Keycloaks standardoppsett ble beholdt. Vi justerte farger og enkelte detaljer i stiloppsettet. Det holdt for vårt behov og det var raskt å sette opp. Vi trengte ikke et stort redesign, bare nok til at det så ut som en del av resten av systemet.

5.1.2 Valg I Betalingssystemet

For Betalingssystemet var det gjort mange valg og avveininger. Vi hadde et klassediagram eller UML (Unit Modelling Language) diagram for å illustrere hvordan betalingssystemet skulle se ut og samhandle i kontakt med tredjeparts bankløsninger, og det allerede eksisterende medlemskapssystemet.

Vi hadde et urealistisk mål om å lage en slags åpen kildekode, lettvekt konkurrent til “Stripe”. For å få til dette var det ønskelig å ha med så mange ulike betalingsløsninger som mulig. EFN ønsket at vi systemet skulle støtte fakturabetaling, betaling med vipps, kontantfaktura, bitcoin, kortbetaling og alt annet vi fikk tid til. Oppdragsgiver skjønte selvfølgelig at vi ikke fikk tid til alt dette og ba oss heller starte med implementasjon av faktura og kontantfaktura og så overføre vipps fra medlemskapssystemet til

betalingsystemet. Betalingssystemet ble designet for at det skal være lett å knytte det til flere ulike betalingsløsninger ved videre utvikling i systemet. Vi innså tidlig at vi måtte ha en database og effektiv løsning for å håndtere betalinger, og for å ha kontroll på utstående betalinger. Tidlig i fasen ble denne løsningen kalt ”den magiske verifikasjonsboksen”, men har over tid blitt en fullintegrert og ferdigstilt løsning.

Push eller Pull løsning

Når vi skulle strukturere betalingssystemet vårt var en av de første vurderingene vi måtte ta stilling til om vi skulle ha en *push* eller *pull* løsning for å koble til eksterne aktører. Si for eksempel at en faktura blir utsendt og venter betaling. I en pull løsning vil da betalingssystemet prompte banken en gang per tidsintervall og sjekke om fakturaen er betalt. I en push løsning vil et tredjepartsprogram ha ansvar for å sende beskjed til betalingssystemet om at fakturaen er betalt. Oppdragsgiver sa at dette var et valg vi måtte ta selv og at det var ingen feil løsning, men at vi måtte ha en god begrunnelse for hvilken løsning vi valgte å gå for.

Fordelene ved en push løsning er at systemet blir oppdatert umiddelbart når det kommer inn en betaling, mens en pull basert system vil måtte vente til neste tidsintervall til å finne ut om fakturaen er betalt eller ikke. En push basert løsning er også mer ressurseffektiv enn én pull løsning, systemet vil ikke trenge å sende forespørsler til banken hvert tidsintervall. Med en push basert løsning vil det heller være et tredjepartssystem som sender betalingssystemet forespørsel ved betalt faktura.

Styrkene ved en pull basert løsning er at du har full kontroll over systemet. Du kan selv bestemme hvor ofte banken skal spørres. Når spørringene kommer fra dit eget system, er det mye enklere å regulere sikkerhet og autentisering. I en push basert løsning er du sårbar hvis leverandørens push systemet er nede eller feiler. Da vil din eneste sikre løsning være å falle tilbake på en pull løsning om noe skulle gå galt.

Vi bestemte oss for å gå for en push løsning. Grunnen til dette er at en push løsning er mye mer skalerbar, og siden det er meningen at betalingssystemet skal være åpen kildekode og brukbart av alle ble dette det åpenbare valget. Det er også lettere å implementere en push løsning og det finnes mange veletablerte push-systemer vi kan ta i bruk for eksempel webhooks. Vi kom også frem til at det kunne være lurt å lage en pull løsning i fremtiden som kunne benyttes dersom det oppsto feil i push løsningen.

XML eller JSON for PDF - generering

Uavhengig av betalingsløsning, så trenger man et kjøpsbevis/kvittering/kontaktfaktura os. Dette har vi generert via det vi kaller «PDF-modulen». Det finnes mange løsninger der ute, og Python har mest sannsynlig flere biblioteker for å håndtere PDF-generering. Vi har hatt en filosofi om at hvis det finnes en open source og enkel løsning fra før av så benytter vi oss av det.

Et av de teknologiske valgene vi skulle ta er hva slags dataformat vi skulle ha. I IT-verdenen så er JSON en veldig etablert standard i dag, og brukes flittig i moderne REST APIer for å kommunisere og overføre data. JSON er en subsyntax for javascript for å kommunisere og sende data, men i dens tidlige dager var det få verktøy og nyttige biblioteker for å trekke ut viktige selektive data, slik som for XML (Šimec & Magličić, 2014).

I Business verden så er det viktige at data trekkes ut og håndteres korrekt, da feil og slike ting kan få alvorlige konsekvenser. Så fra veldig tidlig av i ERP-systemer og regnskapsprogrammer så ble Universal Business Language (UBL) standarden, og denne er bygget på XML. Vi bestemte oss derfor for å importere og benytte oss av XML som standarden for vår modul, slik at dette systemet bedre kan kommunisere og samarbeide med eksisterende systemer og løsninger der ute. Med tanke på at vi bruker FastAPI sine bibliotek og en REST-API-struktur kunne vi sikkert rettferdigjort å bruke JSON, men vi valgte likevel å gå for den etablerte standarden for behandling av finansielle data.

Valg av testbibliotek

Vi hadde også en diskusjon over hvilket testverktøy vi skulle bruke. Initialet så begynte flere i gruppen å benytte seg av ulike bibliotek, slik som unittest og pytest. Begge er veldig anerkjente og vidt brukt blant Python utviklere. Da vi satt oss ned for å velge etter at vi fikk beskjed om at vi måtte forholde oss til ett verktøy, endte vi opp med å gå for unittest. Rett og slett fordi vi hadde kommet lengst med unittest på dette tidspunktet, og at pytest beskrives som litt mer omfattende for nybegynnere å sette opp enn unittest (Taqi, 2024). Dette er fordi pytest er et tredjepartsbibliotek, og unittest er et integrert Pythonbibliotek. Pytest gjør for eksempel automatiske tester, og gir litt mer detaljerte testrapporter, men unittest er litt lettere å lære seg og vi hadde også kommet litt lengre på de testene. Til slutt valgte vi derfor å gå for unittest, ikke nødvendigvis fordi det er bedre, men fordi det virket enklere der og da i øyeblikket. Det hjelper også med beslutningen at vi slipper å laste ned enda et eksternt bibliotek.

5.2 Refleksjon

Vi begynte på prosjektet andre uka i januar og hadde da et oppstartsmøte med oppdragsgiver. På dette møtet ble oppdragsgiver og vi enige om at det ville være vanskelig å definere prosjektets fulle omfang. Dette skyldtes både at oppdragsgiver ikke hadde tidligere erfaring med bachelorstuderter og at vi selv var usikre på hvor mye tid og ressurser et slikt prosjekt ville ta, gitt vår eksisterende kompetanse.

I ettertid har vi reflektert rundt hvorvidt det var en god idé å ha «flytende» eller åpne målsetninger. Her er det litt blandede meninger i gruppen, og fra oppdragsgiver. Det at målposten alltid endrer på seg kan for noen ha påvirket flyt og motivasjon, mens for andre kan det å ha en iterativ tilnærming med ukentlig delmål gitt en bedre flyt og fremdrift.

Selv om vi under prosjektperioden delte oppgaver inn i “must haves” og “nice to have’s”, var det likevel en slags flytende overgang mellom disse to. Dette førte til at ambisjonsnivået i teorien kunne bli uendelig høyt. Hvorvidt en slik tilnærming er det beste for å oppnå gode resultater er noe vi har reflektert over i etterkant.

En annen ting vi har reflektert mye over er oppstartsprosessen og hvilken retning vi startet prosjektet i. De første ukene av prosjektperioden måtte vi bruke mye tid på å sette oss inn i Docker og medlemskapssystemet. Dette gjorde at vi fikk programmert veldig lite og utviklingen gikk sakte.

På det siste møtet med oppdragsgiver hadde vi en felles refleksjons økt, hvor nettopp oppstarten raskt ble trukket frem som et punkt vi ville gjort annerledes dersom vi hadde startet på nytt. Vi brukte kanskje litt for mye tid på å oppdatere avhengigheter og gå gjennom kodebasen for sårbarheter - uten å nødvendigvis vite hva vi lette etter. Det å sette oss inn i Docker og sette opp containere ble også vektlagt kanskje litt for tidlig i prosessen. Vi ble alle enige om at en grundigere kodegjennomgang med forklaringer og raskere start på selve kodingen samt tettere veiledning i denne fasen kunne ha gitt oss et bedre forsprang.

For vår egen del har vi kanskje hatt litt stor terskel for å ettersørre hjelp fra oppdragsgiver, og slik har mye tid gått med på «knoting» på ting som kunne ha blitt løst tidligere. En annen refleksjon vi har gjort oss opp var også det at vi byttet “grupper” 1/3 underveis i løpet (Vi har alltid vært to stykker på medlemskapssystemet, og to på betalingssystemet). Det har i begge grupper vært en person som har hatt litt bedre oversikt over progresjonen og utviklingen, slik

det ofte er i utviklerprosjekter. Ettersom vi grunnet flere omstendigheter var nødt til å jobbe mye individuelt så har det vært vanskelig å bare hoppe på et prosjekt som var godt i gang. Vi hadde alle i gruppen forskjellige arbeidstider på våre deltidsjobber, og mye tid ble dermed brukt på å sette seg inn i hva den andre makkeren hadde gjort, og kodet. Vi tror ideen i utgangspunktet har vært god, men at det skapte litt stopper for flyten.

En annen refleksjon vi gjorde, og som alle parter var enige i var at vi burde hatt flere fysiske samlinger med oppdragsgiver og hele gruppen, og også på et tidligere tidspunkt enn det vi hadde. Vi hadde oppstartsmøte i januar, men etter det så var neste fysiske samling i mars, og dette møtet varte i 8 timer. Vi burde heller ha delt dette møtet i to, og hatt en 4-timers samling i februar. Dette er fordi disse møtene var veldig effektive og ga oss mye erfaring og forståelse, men 8 timer ble litt vel langt.

5.3 Læringsutbytte

Vi har alle blitt veldig mye tryggere på viktige prosessverktøy og bransjestandarder, slik som Git og Github, hvordan vi jobber i Scrum og Kanban samt samarbeid i gruppe. Vi har blitt tryggere på distribuering-løsninger slik som Docker, og god prosjektstyring.

En stor del av arbeidet har gått ut på å videreutvikle et allerede eksisterende system. Det å sette seg inn i og forstå andres kode og systemarkitektur har vært en stor utfordring, men også svært lærerikt. Dette er en viktig ferdighet som man bør mestre i arbeidslivet, hvor utvikling ofte skjer på eksisterende systemer.

Et annet viktig læringsområde har vært sikkerhet, spesielt knyttet til håndtering av sensitiv data. Vi har lært hvordan man kan bruke miljøvariabler for å holde passord, API-nøkler og annen konfigurasjon skjult fra GitHub.

Videre så har vi alle fått mye bedre teknisk forståelse. Python er et språk vi har hatt tidligere i fagene datanettverk og skytjenester og introduksjon til kunstig intelligens. I løpet av denne prosjektperioden har vi virkelig fått lært masse når det gjelder Python og fordelene som kommer med det. Men generelt sett vil vi si at vi også har fått en bedre forståelse for objektorientert programmering, full-stack utvikling og APIer som helhet.

Men det desidert største læringsutbytte er at vi har jobbet i et tilnærmet virkelig arbeidsscenario som vi kan forvente å se i arbeidslivet. Det er det som har gitt oss vårt største læringsutbytte.

5.4 Videreutvikling

Vi anser ikke prosjektet som ferdig. Vi har i tidligere seksjoner på denne rapporten fremhevet flere svakheter og oppgaver som vi kunne tenke oss å fullføre, eller forbedre. Ettersom ingen av oss har erfaringer med moderne utviklingssystemer så har ting tatt lengre tid enn vi kunne forutse. Men vi har derimot gjort mye av grunnarbeidet for et fungerende medlemskaps- og betalingssystem. Vi har også lagt inn innsats for å dokumentere på best mulig måte. Koden skal bli åpen kildekode og alt er blitt dokumentert i Github sin versjonskontroll. Vi har lagt til rette for at andre kan ta over stafettpinnen og fullføre og utbedre medlemskaps- og betalingssystemet.

5.4.1 Betalingssystemet

Betalingssystemet spesielt har en del igjen å gjøres på før det er produksjonsklart. Her går vi gjennom hva vi tenker er de neste viktige endringene som må gjøres i systemet.

Frontend

Frontend siden har vært vanskelig å jobbe med. Dette har vært fordi denne delen av programmet ikke faktisk skal ligge i betalingssystemet. Det har hele tiden vært ment at administrasjonspanelet skal bli flyttet over i medlemskapssystemet. Mange av de tenkte funksjonalitetene i frontend er tenkt å komme ifra medlemskapssystemet, for eksempel er mange funksjoner avhengig av medlemsdata. Vi har ikke fått lov til å koble til medlemskapssystemet før det var klart for produksjon, ettersom at ekstra endringer kun ville utsatt en release.

Den nåværende adminpanelet er brukt hovedsakelig til rask testing av backend og får å se at API-ene virker mellom en react side og backend. Mye er konfigurert med midlertidig data istedenfor data fra medlemskapssystemet. Det neste steget for frontend er å koble til medlemskapssystemet og begynne å integrere data fra medlemsdatabasen. Frontend burde også ha et utseende som matcher EFN sitt tema. Den nåværende siden bruker HMAC autentisering mellom frontend og backend, oppdragsgiver mente det var et noe overdrevent sikkerhetstiltak. En vurdering må gjøres på om dette burde fjernes eller ikke. Per nå åpner admin panelet til en innloggingsside som ikke er stort mer en pynt. Siden burde fjernes og en skikkelig sikkerhetsløsning burde ordnes for frontend ved implementasjon i medlemskapssystemet.

Backend

Backend har kun funksjonalitet for å oppdatere databasen og snakke med PDF-modulen og frontend. Den er bare mulig å styre manuelt gjennom adminpanelet eller manuelle API kall.

Det neste hovedmålet vil være å få koblet til vipps. Vi gjorde et forsøk på å kopiere vipps-systemet som lå i medlemskapssystemet og bruke det i vårt system, dette var en feil. Vipps har en omfattende guide på hvordan det bør integreres i et system. Etter at vipps er integrert kan man se videre på andre betalingsløsninger det er nyttig å ha med, for eksempel kortbetaling eller klarna. Det vil også måtte settes opp et eksternt verktøy, kanskje en webhook som pusher betalingsnotifikasjoner til betalingssystemet. Selv om dette vil være et stykke i fremtiden. Litt mindre endringer som burde gjøres i koden er å implementere database backup som per nå er utviklet i en feature gren, men ikke er merget. Denne burde ha tester som ser at den faktisk tar backup av databasen slik som det ble implementert i medlemskapssystemets database.

PDF

PDF modulen ble nesten ferdigstilt. Den største endringen som mangler er å lage en frontend side der organisasjoner kan legge inn sin informasjon og logo og få tildelt sin egen API-nøkkel. Denne organisasjonsdataen lagres nå i databasen som hører til backend. Det er tenkt at PDF systemet skal være et selvstendig system og det burde tas en vurdering på om det burde opprettes en egen database kun til PDF systemet. En ekstrafunksjon som hadde vært fint å ha er muligheten til å ta imot forespørsler i JSON format i tillegg til XML. Rate limiteren må også fikses før systemet er produksjonsklart.

Vi ser potensiale for at denne åpne kildekodeløsningen med litt videreutvikling kan gi gode samfunnsmessige gevinst for samfunnet som helhet, og da spesielt til alle ideelle organisasjoner der ute som trenger å holde kontroll på medlemslister, kontingenter og betalinger. Dette vil frigjøre midler for EFN og andre organisasjoner der ute til å bruke mer penger på å utgjøre en forskjell i verden.

5.5 Konklusjon

Dette prosjektet har vært en krevende, men veldig lærerik reise der vi har fått muligheten til å arbeide med reelle problemstillinger knyttet til utviklingen av to funksjonelle systemer for en faktisk oppdragsgiver. Våre mål ble delvis oppnådd. Slik systemet er i dag, er det trolig ikke

klart for produksjon, ettersom det fortsatt gjenstår testing og noen utbedringer før betalingssystemet kan tas i bruk.

Medlemskapssystemet er i stor grad ferdigstilt, men den nye betalingsløsningen er foreløpig ikke integrert. Endringene vi har implementert i medlemskapssystemet har blitt pushet fra *staging* til *main*, og vi har hatt en gjennomgang av disse sammen med oppdragsgiver. I denne prosessen har vi fått konstruktive tilbakemeldinger på mindre forbedringspunkter, som vi har rettet opp underveis.

Oppdragsgiver sa at vi var den første gruppen studenter han hadde veiledet og at det var en ny opplevelse for han. I ettertid har han reflektert over prosessen og nevnt flere ting han ville gjort annerledes til en annen gang. Han antydet også at han ønsket en ny gruppe studenter neste år, som kan fullføre det vi startet på i prosjektet vårt. Det er sannsynlig at EFN vil ferdigstille medlemskapssystemet, mens betalingssystemet trolig vil stå urørt i mellomtiden.

6. Referanseliste

Ab—Apache HTTP server benchmarking tool—Apache HTTP Server Version 2.4. (u.å.).

Hentet 20. mai 2025, fra <HTTPS://HTTPd.apache.org/docs/2.4/programs/ab.html>

Dependabot [Github Repotorium]. (2025). GitHub. <HTTPS://github.com/dependabot>

Docker. (2025, april 9). *Docker: Accelerated Container Application Development.* <HTTPS://www.Docker.com/>

Docker Desktop WSL 2 backend on Windows. (2025, april 9). Docker Documentation. <HTTPS://docs.Docker.com/desktop/features/wsl/>

Draw.io. (u.å.). Hentet 21. mai 2025, fra <HTTPS://www.drawio.com/>

FastAPI. (u.å.). Hentet 20. mai 2025, fra <HTTPS://fastapi.tiangolo.com/>

Gowda, P. G. A. N. (2022). *Git branching and release strategies.* <HTTPS://doi.org/10.5281/ZENODO.14221771>

Guide to Agile Practices. (2012, april 29). <HTTPS://web.archive.org/web/20120429172731/HTTP://guide.agilealliance.org/guide/unites.t.html>

Keycloak. (u.å.). *Keycloak.* Keycloak. Hentet 19. mai 2025, fra <HTTPS://www.keycloak.org/>
Mathesar—Open source UI for Postgres databases. (u.å.). Mathesar.Org. Hentet 21. mai 2025, fra <HTTPS://mathesar.org/>

Member Spotlight: EFN. (u.å.). *European Digital Rights (EDRi).* Hentet 12. mai 2025, fra <HTTPS://edri.org/our-work/member-spotlight-electronic-frontier-norway/>

Microsoft. (2025). *Visual Studio Code* [Programvare]. <HTTPS://code.visualstudio.com/>

Ozkan, N., Bal, S., Erdogan, T. G., & Gök, M. Ş. (2022). Scrum, Kanban or a Mix of Both? A Systematic Literature Review. *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*, 883–893. <HTTPS://doi.org/10.15439/2022F143>

PostgreSQL. (2025, mai 21). PostgreSQL. <HTTPS://www.postgresql.org/>

Python Software Foundation. (u.å.-a). *unittest.mock—Mock object library.* Python 3.13.3 Documentation. Hentet 19. mai 2025, fra <HTTPS://docs.Python.org/3/library/unittest.mock.html>

Python Software Foundation. (u.å.-b). *unittest—Unit testing framework*. Python 3.13.3 Documentation. Hentet 19. mai 2025, fra <HTTPs://docs.Python.org/3/library/unittest.html>

Python Software Foundation. (2025, mai 7). *Welcome to Python.org*. Python.Org. <HTTPs://www.Python.org/>

React. (u.å.). Hentet 21. mai 2025, fra <HTTPs://react.dev/>

Heap.io. (u.å.). *What is a Tech Stack: Examples, Components, and Diagrams*. Heap. Hentet 20. mai 2025, fra <HTTPs://www.heap.io/topics/what-is-a-tech-stack>

Reentrancy (computing). (2025). I *Wikipedia*.

[HTTPs://en.wikipedia.org/w/index.php?title=Reentrancy_\(computing\)&oldid=1290051072](HTTPs://en.wikipedia.org/w/index.php?title=Reentrancy_(computing)&oldid=1290051072)

SequenceDiagram.org. (u.å.). *SequenceDiagram.org—UML Sequence Diagram Online Tool*. Hentet 21. mai 2025, fra <HTTPs://sequencediagram.org>

Siapno, N. (2023, juli 24). Performance Testing vs. Load Testing vs. Stress Testing. *Postman Blog*. <HTTPs://blog.postman.com/performance-testing-vs-load-testing-vs-stress-testing/>

Signal Messenger: Speak Freely. (u.å.). Signal Messenger. Hentet 21. mai 2025, fra <HTTPs://signal.org/nb/index.html>

Šimec, A., & Magličić, M. (2014). *Comparison of JSON and XML Data Formats*.

Sommerville, I. (2016). *Software engineering* (Tenth edition). Pearson.

Taqi, M. (2024, mars 15). *Pytest vs Unittest, Which is Better?*

<HTTPs://blog.jetbrains.com/pycharm/2024/03/pytest-vs-unittest/>