Semester: CCT2

Title: Social media platform for IT enthusiasts

Project Period: 03.02.2025 - 28.05.2025

Semester Theme: Complex Data Management and Analysis

Supervisor(s): Lene Tolstrup Sørensen

Project group no.: 4

Members:

Abdirashid, Ibtisam Abdihaq

Burhenne, Malte Bach

Jensen, Lucas Ta

Kleist, Stinnia Thala Louisa Hansen

Lindau-Skands, Sebastian

Rashed, Jumana Thammer Shakir

Pages: 87

Finished: 27-05-2025

Aalborg University Copenhagen
A.C. Meyers Vænge 15 2450
København SV

Semester Coordinator: Lene Tolstrup Sørensen

Secretary: Nele Pernille Staun Hvid

Abstract:

This project researches and implements a social media platform based on the interests of IT enthusiasts, and how to ensure the data safety and anonymity of the users. It addresses the importance of developing a proper system to safely secure datastoring, the importance of working with frontend and backend development. Through litterature studies, proper planning of requirement specifications and diagrams, as well as user tests it demontrates how data can be collected and stored securely without compromising the safety of the users. The prototype is presented in the form of a website, and the limitations of the project are discussed and disclosed.

Indholdsfortegnelse

1 Introduktion

Sociale medier er en fast del af vores hverdag. Vi bruger dem til at kommunikere, dele viden og holde os opdateret, men der er også udfordringer, især når det handler om anonymitet og privatliv. Mange platforme samler store mængder data om brugerne og giver ikke altid mulighed for at færdes anonymt.

Vores mål har været at udvikle en prototype på en platform hvor teknologientusiaster kan mødes i en tryg online platform. Vi har arbejdet med at designe funktioner der tager højde for anonymitet, datasikkerhed og gode brugeroplevelser. Samtidig vil vi gerne skabe rammer for at brugerne selv kan præge indhold og debat. Projektet er udviklet inden for temaet *Complex Data Management and Analysis* og derfor har vi også fokuseret på hvordan man strukturerer og håndterer data på en effektiv og sikker måde. Det gælder både i forhold til platformens tekniske opbygning, og hvordan vi respekterer brugernes ret til privatliv.

Derfor har vi valgt at udvikle platformen GNUF. Navnet GNUF stammer fra en kombination af det open source software initiativ GNU, og Forum. GNU som startede som et initiativ for at gøre UNIX [1] mere desktop venlig, og nu er blevet kombineret med Linux i GNU/Linux, som er en række software til håndtering af en computer, heriblandt systemD (Afsnit 8.4), desktop environments, og mere.

2 Problemstilling

Brugernes sikkerhed og anonymitet kommer mere og mere i fokus i takt med den stigende overvågning gennem AI og algoritmer. Samtidig bliver det sværere for den enkelte bruger at udtrykke sin identitet i en teknologisk verden domineret af foruddefinerede strukturer og begrænsede designmuligheder.

Hvordan kan vi skabe en platform målrettet IT-entusiaster, hvor brugerne kan interagere anonymt og trygt, med fokus på organiseret datahåndtering?

3 Rapportens struktur

Projektet indledes med en introduktion til emnet samt en præsentation af problemstillingen. Derefter redegøres der for de metoder, der er blevet anvendt til at forme projektets forløb. Dette omfatter blandt andet en litteratursøgning, projektledelse gennem agile metoder, udførelse af forskellige brugertests samt udviklingen af den endelige løsning. Herefter følger et afsnit med relevant baggrundsviden, herunder om GDPR og anonymitet/privatliv, som danner grundlag for forståelsen af det følgende afsnit. State of the Art afsnittet undersøger, hvordan andre har arbejdet med lignende problemstillinger, hvilke metoder og værktøjer der er blevet anvendt, og har givet gruppen indsigt i, hvilke krav der bør stilles til projektet. Det følgende afsnit beskriver hvordan projektets designløsning er udtænkt og udført, blandt andet gennem læring fra State of Art, systembeskrivelse, userstories, forskellige diagrammer og kravspecifikationer samt selve implementeringen af systemet – herunder en beskrivelse af de vigtigste funktioner. Den efterfølgende afsnit fokuserer på testningen af systemet. Her vurderes det, om kravspecifikationerne er blevet opfyldt, og resultaterne af testene beskrives og diskuteres kort. Til sidst diskuteres projektets løsning, med udgangspunkt i en vurdering af det endelige produkt og gruppens tilgang til problemstillingen. Hele projektet afrundes i en konklusion, som sammenfatter de væsentligste resultater og refleksioner.

4 Metoder

I dette afsnit beskrives metoderne, som gruppen har anvendt i projektets forløb. Metoderne omfatter blandt andet litteratursøgning for at danne det faglige grundlag, projektledelse for at sikre en struktureret arbejdsproces og brugertests til at evaluere prototypens brugervenlighed samt udviklingsprocessen af projektets løsning. Metoderne har sikret en god gruppesamarbejde, samt sikret en målrettet tilgang gennem hele projektet.

4.1 Litteratursøgning

For at skabe et godt og dækkende "State of the Art" afsnit valgte vi i gruppen at arbejde tæt sammen gennem hele processen. Som det første skridt fandt hver af os 2-3 relevante kilder, der kunne være interessante i forhold til vores emne. Derefter mødtes vi og gennemgik alle kilderne sammen, hvor vi diskuterede deres indhold, relevans og falige kvalitet. På den måde kunne vi i fællesskab udvælge de mest brugbare kilder til videre arbejde som en fællesbeslutning.

Vi udarbejdede vores søgninger ved at benytte flere anerkendte databaser og søgeplatforme, såsom AAU's PRIMO, Google Scholar og IEEE Xplore. I alt fandt vi cirka 10–12 kilder, som vi sammen gennemgik for at vurdere relevans, kvalitet og aktualitet. Vi anvendte søgeord som "sociale medier engagement", "brugeradfærd online", "digital kommunikation" og "privatliv på sociale medier". Disse nøgleord hjalp os med at identificere de kilder, der bedst kunne belyse de emner, vi ønskede at dække, og sikrede, at udvælgelsen af kilder foregik metodisk og grundigt.

Efter udvælgelsen fordelte vi kilderne mellem os, så hver person fik ansvaret for at skrive et kort afsnit på cirka en halv til en hel side om én af kilderne, evt. med tilføjelse af et relevant billede. Her beskrev vi både, hvad kilden handler om, og hvordan den er relevant i forhold til gruppens projekt. Da alle havde skrevet deres del, satte vi os sammen og gennemgik teksterne i fællesskab. Vi læste hinandens afsnit, gav kommentarer og foreslog ændringer, så indholdet blev mere klart, og sproget blev mere ensartet. Vi havde fokus på både faglighed og sammenhæng i teksten og rettede løbende til, indtil vi var tilfredse med helheden. Ved at arbejde sammen på den måde har vi fået et godt overblik over, hvad der allerede findes af viden, som vi kan bygge videre på i resten af rapporten.

4.2 Agil projektledelse

I løbet af projektet har vi arbejdet ud fra en agil tilgang, hvor vi hele tiden har tilpasset os og fordelt opgaver efterhånden. Vi startede ikke med en fast plan, men fandt naturligt ud af, hvem der tog sig af hvad, efterhånden som projektet skred frem. Vi har mødtes regelmæssigt for at snakke om, hvor langt vi var nået, hvad der manglede, og hvordan vi skulle komme videre. Vi har brugt en iterativ process igennem projektet, for at sikre et godt endeligt produkt, hvor vi fokuserede på en bestemt del ad gangen, eksempel design, kodning eller test og derefter gik videre til den næste del. Det gjorde det nemt at rette fejl og tilpasse ting undervejs, i stedet for at vente til sidst.

User stories har også været en vigtig del af arbejdet. De hjalp os med at finde ud af, hvilke funktioner systemet skulle have, og hvad brugerne skulle kunne. Det gjorde det nemmere at få et klart overblik over, hvordan produktet skulle udvikles. Herefter har vi lavet både funktionelle og ikke-funktionelle krav. De har hjulpet os med at sætte retningen af projektet og

prioritere, hvad der var vigtigt for systemets funktioner samt brugervenlighed og performance af det udviklede platform. Alt i alt har det agile givet os en god måde at holde styr på projektet og arbejde sammen på en fleksibel måde.

Samtidigt har vi gjort brug af flere diagrammer for at kunne danne et bedre overblik over hvordan systemet skulle sammensættes. Vi lavede context diagrammer for både frontend og backend, som viste, hvordan brugere, admin, browser og database spiller sammen. Derudover brugte vi use case-diagrammer til funktioner som eks. login, opslag og communities, for at vise hvad brugeren skal kunne gøre i systemet. Data flow-diagrammerne hjalp os med at forstå, hvordan information bevæger sig mellem systemets dele, og klassediagrammet gav os struktur over backendens opbygning med controllere og modeller. Vi lavede også et ER-diagram, som viser, hvordan dataelementer som brugere, opslag og communities hænger sammen i databasen, f. eks. at en bruger kan lave flere opslag eller være med i flere communities. Diagrammerne har været en vigtig del af den agile proces og har gjort det lettere at planlægge, udvikle og teste løsningen.

4.3 Projektledelse/værktøj

Til arbejdet med rapporten har vi valgt at bruge Typst [2] som vores primære værktøj. Vi valgte Typst, da det gør det nemt at lave en flot og overskuelig opsætning, og samtidig er det let at samarbejde i. Typst er en open-source system baseret på markup, som er nem at lære og danner et bedre overblik over lange dokumenter, såsom rapporter, og flere personer kan arbejde på dokumentet på samme tidspunkt. Det har været en stor fordel, at vi alle har kunnet arbejde i de samme dokumenter og se ændringerne med det samme, hvilket har gjort skriveprocessen mere effektiv og mindre forvirrende. Ud over selve rapporten har vi også oprettet andre dokumenter i Typst, som vi har benyttet til at holde styr på hele projektforløbet. Et af dokumenterne (fra bilaget ***Procesanalyse***) har vi brugt til notetagning fra vejledermøderne samt spørgsmål, som vi har haft undervejs. Det har hjulpet os med at samle vigtig feedback og følge op på de punkter, vi har talt om med vejlederen. Et andet dokument (fra bilaget ***Procesanalyse***) har fungeret som vores fælles planlægnings- og mødedokument. Her har vi skrevet, hvornår vi skulle mødes, hvad vi skulle nå til hvert møde, og hvem der havde ansvar for hvad. Vi har også brugt det til at fordele individuelle opgaver og skrive opsummeringer af, hvad hvert gruppemedlem har arbejdet med mellem møderne. At have det hele samlet i Typst har gjort det nemt at holde overblik, og det har givet os en struktureret og samlet måde at arbejde på, både med indholdet af rapporten og med selve samarbejdet.

Programmeringen er foregået på gruppemedlemmernes foretrukne udviklingsmiljøer, og alle forskellige dele af kodningen er blevet gemt og løbende opdateret på GitHub. Github er en cloud-baseret platform, hvor man kan lagre, dele og arbejde sammen om programmeringen, gemt i repositories man selv opretter [3]. Gruppens primære GUI-værktøj har været GitKraken til at interagere med de Git baserede repositorier. Dette værktøj har gjort det nemmere at holde overblik over de mange linjer kode samt sørge for at der ikke var for mange merch konflikter. Det er også gennem GitHub, de første udkast til eksempelvis kravspecifikationer, hvilke teknologier gruppen skulle benytte og brainstorm-sessionerne er blevet skrevet ned.

Selve koordineringen af samarbejdet er foregået på Discord. Discord er en gratis beskedapp, hvor man kan skrive sammen over tekst, video eller stemmeoptagelser [4]. Kommunikation omkring opgaver, changelogs samt fravær ved sygdom eller forsinkelser er foregået over en

dedikeret Discord server. Her har gruppen lavet flere kanaler til at kommunikere omkring forskellige dele af projektet, for eksempel en kanal for frontendudviklingen, opgaver lavet i undervisninger, idéer til P3 projektet samt en generel kanal for overordnede ting som opdatering til rapporten, sygemeldinger eller hvornår gruppen mødes næste gang. Dette er lavet parallelt med gruppens gruppemøde skabelon på Typst.

Gruppen har også benyttet Figma til at lave tidlige skitser af hjemmesiden, før udviklingen af programmeringen startede. Figma er et samarbejdsbaseret, webbaseret design- og prototypeværktøj, der primært bruges til at skabe brugergrænsefladedesign (UI) og brugeroplevelsesdesign (UX)[5]. Her er det første udkast til hjemmesidens udseende blevet skitseret, og efterfølgende implementeret under programmeringsudviklingen.

Til selve programmeringen af produktet har vi benyttet Webstorm/Zed til frontendudviklingen og Rider/Zed til backendudviklingen. Webstorm og Rider er IDE'er (integrated development environment) som er lavet af Jetbrains med fokus på individuelle programmeringssprog hvor Rider har fokus på C# og Webstorm har fokus på web development. Zed er en rust baseret IDE som giver en minimalistisk og overskuelig approach til kodning, mens stadig at have kraftfulde funktioner såsom tiling windows, AI assistance etc.

4.4 Brugertests

For at få et indblik i hvordan rigtige brugere oplevede platformen, valgte vi at gennemføre en brugertest. Eftersom applikationen primært er drevet af brugeraktivitet, vil en brugertest være den mest effektive måde at teste den på.

Med en brugertest kan det undersøges om applikationen er brugervenlig for personer uden tidligere erfaring med den. Brugertestning kan også anvendes til bedre at forstå brugeradfærd, og hvordan brugere naturligt interagere med hjemmesiden. Hertil kan der også findes fejl som kan have været overset under intern testning af udviklerholdet, og der kan også indsamles feedback fra deltagerne om deres oplevelse, og hvilke ting de føler der manglede. Det bidrager også til den iterative proces, i at det giver værdifuld input til iterative designforbedringer.

I GNUF blev der anvendt forskellige former for brugertestning for at belyse forskellige aspekter af hjemmesiden. Dette vil yderligere uddybes i Afsnit 9.

4.5 Udvikling af løsning

Udviklingen af løsningen til vores sociale platform GNUF har været en iterativ proces præget af agil projektledelse, hvor vi løbende har tilpasset vores arbejde ud fra feedback, test og nye erkendelser. Vi har fokuseret på at bygge en platform, der understøtter brugerinvolvering, fællesskab og sikkerhed med et moderne og brugervenligt design. Projektet tog udgangspunkt i en række user stories og kravspecifikationer, som dannede grundlaget for både design og implementering. De funktionelle og ikke-funktionelle krav blev udviklet på baggrund af litteratursøgning og analyser af lignende platforme såsom Reddit, samt med udgangspunkt i modeller som SMPM og COBRA (Afsnit 6.2). Disse indsigter hjalp os med at identificere centrale behov og forventninger blandt brugerne.

Vores udviklingsproces var opdelt i faser med fokus på design, kodning og test. Frontend delen blev udviklet i React og TypeScript, hvor vi anvendte biblioteker som TailwindCSS og Shadcn. Valget af TypeScript blev truffet pga. dets statiske typer og forbedrede fejlhånd-

tering. Webapplikationen blev opbygget med fokus på komponentstruktur, brugeroplevelse og dynamisk indhold. Backend delen blev udviklet i C# med .NET frameworket. Her byggede vi et API, der håndterer logik for brugere, opslag, kommentarer, communities og interaktioner. Vi udviklede controllere og modeller til at sikre en klar og modulær struktur. Databasen blev udviklet baseret på et detaljeret databaseskema, som understøtter platformens funktioner og skalerbarhed. I hele udviklingsfasen anvendte vi forskellige UML- og diagrammer, herunder use case-, sekvens- og klassediagrammer, for at visualisere systemets komponenter og interaktioner. Dette bidrog til at sikre, at løsningen hang sammen med vores oprindelige design og kravspecifikationer. Afslutningsvis blev løsningen testet med fokus på funktionalitet og performance, og vi brugte brugertests til at identificere eventuelle forbedringspunkter. Den iterative tilgang gjorde det muligt for os at reagere hurtigt på udfordringer og sikre en mere robust og brugervenlig løsning.

5 Baggrundsviden

5.1 GDPR

General Data Protection Regulation (GDPR) er en EU-lov, der regulerer, hvordan persondata håndteres for at beskytte individers privatliv. Loven stiller krav til samtykke, databehandling og brugernes rettigheder. I denne rapport tager vi en række relevante GDPR-artikler i betragtning, selvom vi ikke nødvendigvis følger dem fuldt ud, da det ikke er et krav for projektet. For at sikre lovlig behandling af data kræver Art. 6, at der findes et gyldigt grundlag, fx brugerens samtykke. Art. 7 understreger, at samtykke skal være frivilligt og informeret, og at brugeren til enhver tid kan trække det tilbage, hvilket kan betyde sletning af konto og data**{1}**. Når det gælder beskyttelse af børn, kræver Art. 8, at personer under 16 år skal have samtykke fra en værge for, at deres data kan behandles. Samtidig fastslår Art. 15, at brugere har ret til at vide, hvilke data der indsamles, hvem der har adgang, og at de kan anmode om en kopi af deres data. Selvom vi ikke nødvendigvis implementerer alle disse regler fuldt ud, har vi taget dem i betragtning for at sikre en ansvarlig håndtering af data i projektet.

5.2 Anonymitet Og privatliv

Anonymitet spiller en central rolle i vores projekt, da brugerne forventes at kunne deltage i Communities og dele indhold uden nødvendigvis at afsløre deres identitet. I takt med at digitale platforme bliver mere udbredte, er spørgsmålet om hvordan man beskytter brugerens privatliv og anonymitet, blevet særligt aktuelt. I vores kontekst betyder anonymitet, at brugere kan oprette og interagere med indhold uden at deres virkelige identitet knyttes til profilen eller opslaget. Det er dog vigtigt at understrege, at anonymitet aldrig er absolut på et socialt medie og at det afhænger af, hvordan brugeren integrerer med systemet, hvilke oplysninger de selv vælger at fremvise, og hvordan disse håndteres. Derudover er emnet relevant for flere typer IT-specialister, som kunne være interesserede i projektet.

6 State of Art

Dette afsnit giver oversigt over den nyeste teknologiske udvikling indenfor sociale medier, brugernes online færden samt brugerbeskyttelse. Vi analyserer eksisterende løsninger og metoder til at beskytte brugernes privatliv samt de udfordringer, som brugere og platformudviklere står overfor i en digital tidsalder. Formålet er at skabe et solidt grundlag for forståelsen af aktuelle tendenser og problematikker, der påvirker sikkerheden og adfærden på sociale medier.

6.1 Sociale medier

I artiklen fra socialmedier.dk [6] beskrives de sociale medier som online platforme, hvor brugere kan publicere tekster, billeder, videoer og links samt kommunikere med hinanden ved private beskeder. En stor fordel ved de sociale medier er, at de gør det nemt at kommunikere og engagere sig med andre. De adskiller sig fra traditionelle medier, såsom aviser og nyhedssider, ved at indholdet primært skabes og deles af brugerne selv. Ved at like, dele og kommentere på indhold skabes der livlige online fælleskaber, som er blevet en vigtig del af hverdagen for mange. Samtidigt er platformene blevet vigtige værktøjer for virksomheder, som benytter disse for at reklamere og markedsføre deres services og produkter. Virksomhederne bruger de sociale medier som kilder til markedsføringsindsigt, hvor de observerer, analyserer og forudsiger kundernes adfærd for at opnå konkurrencefordel og overlegen ydeevne [7].

En artikel fra Science Direct [8] understreger dog den mørke side af de sociale medier, som inkluderer cybermobning, afhængighed, falske nyheder samt misbrug af privatlivets fred. Brugerne har en tendens til at ignorere de kort- og langsigtede konsekvenser og potentielle risici ved deres valg, som blandt andet kan føre til øget niveau af angst, søvntab og depression ved stort forbrug af de sociale medier. Hvor det er blevet markant nemmere at komme i kontakt med venner og familie, som bor langt væk, er det også blevet nemmere at miste kontakt med de personer, som sidder i samme rum. For eksempel foreslår udtrykket "shallowing hypothesis" [8] den voksende tendens i hurtige og overfladiske tanker, frem for almindelig daglig reflekterende tænkning. Med andre ord kan et højt forbrug af sociale medier føre til en tendens til overfladisk tænkning, da korte og letfordøjelige informationer dominerer indholdet online. Dette kan påvirke både kognitive og etiske overvejelser.

Derfor er det vigtigt at danne en forståelse på hvordan man kan sikre brugeroplevelsen samt beskytte brugernes private data. En artikel fra AIContentfy [9] beskriver vigtigheden ved at definere kerneegenskaber og funktionaliteter ved sociale medier, især hvad brugerne værdsætter mest, såsom nyhedsfeeds, beskeder og indholdsdeling. En brugervenlig grænseoverflade er afgørende for platformens success, og artiklen [9] understreger vigtigheden ved at benytte frameworks, såsom React eller Angular, som kan forbedre brugeroplevelsen. Derudover beskrives det, at en ordentlig backend-udvikling og en veldesignet database kan sikre effektiv datahentning og skalerbarhed. Artiklen [9] nævner Node.js, Ruby on Rails og Django som mulige værktøjer. En nøje planlagt og designet backend struktur vil danne grundlaget for en robust og skalerbar social media platform.

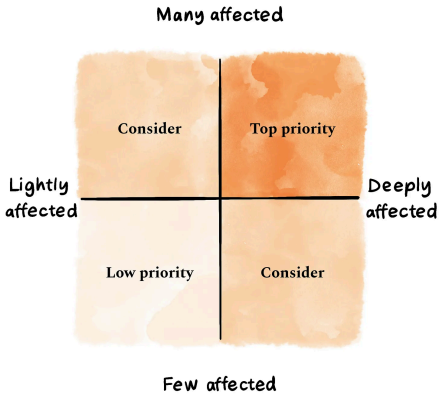
Selvom de sociale medier gør det lettere at forbinde mennesker, rummer de også en risiko for spredning af falske og potentielt skadelige oplysninger. Derfor er det vigtigt at anvende sociale medier med omtanke og en kritisk sans for at undgå faldgruber som misinformation

og afhængighed og tab af dybere refleksion. Samtidigt skal man som udvikler sikre brugernes oplevelser og sikkerhed online ved at udvikle robuste og pålidelige platforme.

6.2 Fokus på brugere

En platform, der er afhængig af brugernes engagement, bør også overveje brugernes behov nøje. En artikel fra Lenny's Newsletter [10] skriver om platformen Reddit og beskriver hvordan forholdet mellem platformudviklere, moderatorer af forummer og deres brugere fungerer. Reddit er et socialt medie med en forumlignende struktur, der dagligt tiltrækker millioner af besøgende. Platformen fungerer som et online fælleskab, hvor brugere kan oprette og deltage i diskussioner om forskellige emner. Platformen er opdelt i mindre subgrupper, kaldet subreddits, som har deres egne fokusemner, såsom teknologi, mad, sport, politik osv. Platformen er i høj grad afhængig af sine moderatorer til at administrere og opretholde forummerne samt facilitere de aktive diskussioner [11].

Tilbage i 2015 gik det galt for platformen, da moderatorerne gik på strejke fordi de følte at deres behov ikke blev mødt af Reddit [10]. Dette medførte et lederskift hos Reddit, og der blev igangsat en proces for at opbygge et system for et stærkere brugerfællesskab. Reddit har udarbejdet et system for at karakterisere feedback baseret på hvor stor en del af brugerne påvirkes af ændringen og hvor højt den burde prioriteres, som ses på Figur 1. For det første skulle man identificere og måle brugernes tillid til platformen. Hos Reddit ser mange brugere platformen som deres eget fællesskab, hvilket betyder at de kan være meget skeptiske over for ændringer, især hvis de strider imod brugernes interesse. Reddit bevarer tilliden ved at være gennemsigtige med brugerne omkring deres planer for mulige ændringer og dermed inddrage brugerne tidligt i udviklingsprocessen.



Figur 1: Reddit feedback assesment. [10]

De andre trin i processen sætter fokus på vigtigheden ved at lytte til de rigtige stemmer i fælleskabet, danne rådgivende råd samt handle hurtigt. De fleste mennesker giver ikke feedback på velfungerende funktioner, men vil være hurtige til reagere på funktioner, som de synes er dårlige. Der findes også mange brugere, som er mere synlige end andre. Det kan for eksempel være dem, som råber højest op, hvor man fejlagtigt kan tænke, at det må være disse brugere som man skal lytte til først. I artiklen [10] beskrives det, at udviklerne havde lyttet til disse stemmer engang imellem, og fundet ud af at deres ønsker tit ikke stemte med resten af brugernes ønsker. For eksempel havde man lavet nogle funktioner for disse højtråbende stemmer, og efter implementering fundet ud af at de nye funktioner ikke var ønsket af resten

af brugerne. Derfor er det vigtigt at formere et rådgivende råd for at have en produktiv diskussion. Normalt består disse råd af en gruppe passionerede fællesskabsrepræsentanter, der skal diskutere og udtænke løsninger udenfor det offentlige øje. Det er også derfor vigtigt at udskifte det her råd løbende for at holde synsvinklerne friske. Til sidst beskrives det i artiklen [10] vigtigheden ved at handle hurtigt ud fra det feedback man gavner fra processen.

For at navigere i de forskellige platformer, både som udviklere og som virksomheder, er det derfor væsentligt at lære brugernes adfærd at kende. Et journal fra ProQuest [12] undersøger, hvordan forskellige former for indhold på sociale medier påvirker brugernes engagement, med særlig fokus på, hvordan både passivt og aktivt engagement påvirkes af indholdstypen. Dette er vigtigt for virksomheder, der ønsker at optimere deres sociale mediestrategier. Studiet analyserer data fra 12 vinmærker over 12 måneder på Facebook, og bekræfter, at rationelle budskaber, såsom informations- og belønningsindhold, fremmer aktivt engagement som likes og delinger, mens emotionelle budskaber, såsom underholdende og relationelt indhold, primært øger passivt engagement, hvor brugere ser og klikker uden aktiv interaktion. For at komme frem til dette anvender forskerne "Uses and Gratifications Theory (UGT)" og "Dual Processing Theory" til at kategorisere social media indhold. Teorien klassificerer disse behov i kategorier som affektive, kognitive, personlige, integrerende og spændingsfrie, som understreger den aktive rolle brugerne har for at forme brugen af de sociale medier. UGT udvikler sig kontinuerligt ved at reagere på kritik og danne en bredere forståelse af brugernes adfærd [13]. Det beskrives hvordan intuitive og rationelle beslutningsprocesser adskiller sig. Indhold, der giver konkret information, som nyheder eller analyser, aktiverer System 2 (rationel tænkning), mens indhold som memes eller personlige historier primært aktiverer System 1 (hurtig, intuitiv tænkning) [12]. Kategorierne inddeles således:

- Informationsindhold skabte både passivt og aktivt engagement – især delinger og likes
- Belønningsindhold, som indeholdte konkurrencer og rabatter, førte til flere likes og delinger, men ikke nødvendigvis til flere kommentarer
- Underholdningsindhold resulterede primært i passivt engagement
- Relationelt indhold skabte også primært passivt engagement

Resultaterne indikerer, at virksomheder bør fokusere på rationelle budskaber, såsom produktinformation og belønningsindhold, for at fremme aktiv interaktion på sociale medier. Disse budskaber er mere effektive til at skabe aktive interaktioner end følelsesmæssige budskaber. For eksempel kan virksomheder implementere kampagner, der kombinerer informative opslag med incitamenter for at øge engagement. Markedsførere bør tilpasse indholdet til målgruppens motivationsfaktorer.

I journalet fra ResearchGate [14] analyseres eksisterende forskning om engagement på sociale medier. Studiet gennemgår tidligere undersøgelser på området og identificerer områder, hvor der er forskningshuller, der kræver yderligere undersøgelser. Journalet beskriver blandt andet engagement som en "multidimensionel" og "polysemisk" proces, hvilket vil sige, at det kan manifestere sig på mange forskellige måder og variere afhængigt af konteksten. Journalet [14] inddrager COBRA-modellen (Consumer Online Brand-Related Activities), som opdeler brugeres engagement i tre forskellige niveauer baseret på hvor høj grad det bidrager til indholdet af det sociale medier. Det første niveau er "Consumption", som omfatter brugere,

der udelukkende forbruger indhold uden selv at interagere aktivt. Dette kan blandt andet være at se billeder eller læse tekster. Det andet niveau er "Contribution". Dette inkluderer brugere som aktivt interagere med indhold, heriblandt at like opslag eller dele det med andre. Det sidste niveau er "Creation", hvor brugere skaber deres eget indhold på de sociale medier, for eksempel ved at oploade billeder, eller lave deres egne opslag. Det inkluderer også at kommentere på andre opslag.

I journalet beskrives det også at der er forskellige metoder hvorpå engagement kan måles. Dette inkluderer kvantitative og kvalitative metoder. Kvantitative målinger er de simpleste, og indebærer for eksempel hvor mange likes, kommentarer og delinger et opslag eller kommentar modtager. Kvalitative metoder derimod går mere i dybden. Et eksempel på sådan en metode er "normalised indexes", som forsøger at måle det engagement noget indhold genererer, sammenlignet med antallet af personer det er blevet vist til. Derved kan man få det gennemsnitlige engegement for hver bruger ved at måle hvor meget engegement forskelligt indhold genererer.

I disse artikler fokuserer man på brugernes interaktion, da et platform ikke kan køre uden brugerne og et virksomhed ikke kan fungere uden at vide hvordan kunderne reagerer på markedsføring i de forskellige sociale medier. Derfor er det afgørende at forstå brugerne og tage deres feedback seriøst, samtidig med at man forholder sig kritisk til de mange stemmer online – både de dominerende og de mere afdæmpede, som ofte overses. Et forkert skridt kan nemt føre til massetab af brugere og kunder. De sociale medier udgør et konstant voksende landskab, hvoraf vigtigheden ved at udvikle sig med tiden ikke kan tilsidesættes. En grundig brugertest i udviklingsfasen samt løbende tests af platformen kan øge brugernes tillid, hvilket styrker platformens chance for succes fra starten og gør det lettere at følge med den løbende udvikling inden for webteknologier.

6.3 Beskyttelse af brugernes privatliv

I takt med den stigende brug af sociale medier er brugernes privatliv blevet et vigtigt emne. Deling af personlige oplysninger, datatracking og algoritmisk overvågning rejser spørgsmål om sikkerhed, kontrol og etiske udfordringer. Et andet journal fra ResearchGate [15] har undersøgt emnet og kommet med et forslag til hvordan den individuelle bruger kan beskyttes. SMPM (Social media privacy model) bruges som forslag til ændring af hvordan privatlivet opfattes i en digital kommunikation. Den traditionele model lægger særlig vægt på, at individet skal have kontrol over sine personlige oplysninger. Det definerer privatliv som et individs evne til at kontrollere adgang til egen personlig information. Den nye model foreslår, at privatliv i sociale medier skal være mere bygget på kommunikation, tillid og sociale normer, frem for kontrol alene. På grund af sociale mediers indbyrdes forbundne natur oplever brugere ofte vanskeligheder med at opnå kontrol, især inden for disse fire områder: anonymitet, redigerbarhed, associationer og varighed. Ved anonymitet vil brugeren gerne gemme sin identitet, af varierende grunde, men mediet vil gerne kunne have disse informationer sporbare. Brugeren vil også gerne kunne redigere indhold før deling. Ved associoationer linker de sociale medier brugerne med andre og anbefaler andre brugere, hvilket begrænser personlig kontrol i, hvem der eventuelt ser ens egen profil eller indlæg. Og ved varighed betyder det, at indlæg forbliver for evigt, som betyder at selvom brugere kan slette deres egne indlæg, vil reposts stadig eksistere.

Tillid og sociale normer fungerer som en erstatning for kontrol, når denne ikke er tilgængelig. De opstår, når brugeren engagerer sig i intentionelle kommunikationer med virksomheder og medier for at forhandle privatlivets betingelser. SMPM er struktureret omkring fire grundprincipper:

1. Privatliv er interdependent - det formes af relationer og tillid, i stedet for udelukkende at være baseret på individuel kontrol
2. Privatliv kan ikke garanteres gennem kontrol alene – sociale mediers struktur og funktioner gør kontrolbaseret privatliv umuligt at opretholde
3. Kommunikation er den primære metode til at regulere privatliv
4. Tillid og normer fastsætter forventninger til privatliv – over tid vil brugere afhænge af etablerede normer og institutionel tillid for at forhandle privatliv

Modellen foreslår et paradigmskifte fra "privatliv som kontrol" til "privatliv som kommunikation". Juridisk skulle lovgivningen fokusere mere på transparens, ansvarlighed og informering af brugeren, således at kommunikative værktøjer kan anvendes i stedet for rene kontrolmekanismer. Sociale medier bør derfor integrere dynamiske privatlivsfunktioner som tilpasser sig med tiden med brugerinteraktioner og sociale normer.

Dette betyder, at et hav af brugerinformationer konstant flyder rundt online, og man har arbejdet længe og stadig arbejder på at finde struktur, som også skal beskytte brugernes privatliv. Et andet journal fra ProQuest [16] beskriver implikationerne ved langsigtet bevaring af API'er (Application Programming Interfaces), herunder de opståede problemer når API'er bruges som den primære måde at udtrække brugerdata fra forskellige platformer. En API fungerer som et kommunikationssystem mellem softwareapplikationer og muliggør adgang til data eller funktioner fra andre systemer uden behov for kendskab til deres interne implementering. Adgang til data med API'er er styret af de privatejede platforme og deres vilkår for serviceaftaler med deres brugere.

Figur 2 beskriver hvordan forskellige typer af API brugere kan tilgå data. En "developer researcher" vil for eksempel have behov for at hente og analysere data for at forme en ny forståelse af brugerne og de underliggende algoritmer, hvor "developers" skal have adgang til data for at opnå en form for forretningsmål og udvikle flere værktøjer til selve platformen og generere økonomisk udbytte. API'er har forskellige funktioner afhængig af platformens formål. For eksempel sørger en social media API for at bruge brugernes informationer til at matche med andre i dating sider, modsat en content API for nyhedssider, som tillader at promovere de nyeste nyheder i de sociale medier. API'er er en stor del af infrastrukturen af det sociale netværk, som driver brugernes oplevelser af adskillige platforme og apps.

Types of API users	Conditions of access and motivations
Account holders are content creators that use the social media platform	• Create user data for APIs • Give developers permission to access and use personal information through the API • Access the API to retrieve account data for personal digital archives
Developers are third-party data brokers who access user data to create new technologies or apps, to advertise content on the platform, or to build data profles to develop technology outside of the platform	• Create tools, clients, and plugins on the platform • Create advertising and promoted content to direct to users, contributing to flter bubbles • Use the API to achieve some kind of business goal through the platform itself, platform development/ data enabler, and source • Motivated by proft generation
Developer researchers are third parties who collect data to investigate, collect evidence, and document social media phenomena, ranging from academics, journalists, civil society groups	• Access API to get more insights into how users communicate, read, consume content, and create social ties • Collect and analyze API data to make claims, to inform decisions, to create new knowledge, to inform policy and regulation • Access the API data to further understand the platform, underlying algorithms, code, data structures, in order to reverse engineer the system itself
Developer stewards are web archives, community archives, or institutional repositories who leverage APIs to collect social media data for preservation purposes and future use	• Produce stable, understandable facsimiles of platform content (get content of the platform for secondary use, research, preservation) • Want to know about the platform, data structure, format, content, context, form • Use the API for reliable, authentic data and to document actions and enforce accountability • Intend to provide long-term access

Figur 2: Types of API users detailing the varying conditions of access and motivations to social media [16]

Hvor en API tillader kommunikation mellem forskellige platforme, skal al data genereret på tværs af disse platforme lagres et sted, og i et hav af muligheder beskriver en anden artikel fra ACM [17] SQLite som den mest udbredte implementerede database i eksistens. I dag findes SQLite i mange enheder, såsom mobiler, computere, webbrowsere, tv og mange andre. SQLite er en open-source relationsdatabasemotor skrevet i programmeringssproget C, gemt i en enkelt fil, som understøtter tusindvis af transaktioner i sekundet. På grund af dens stabilitet, bærbarhed og pladsbesparelse benyttes SQLite eksempelvis til deling af datasæt. Artiklen [17] sammenligner SQLite med med andre transaktionele værktøjer. SQLite's effektive forespørgsels planer og håndtering af datasæt er mere velegnet til større projekter, som kræver mere lagringsplads. Arkitekturen af SQLite's udførelsesmotor er i stand til at identificere hurtigt, hvilke virtuelle instruktioner som er ansvarlige for at sænke programmets ydeevne. Dette muliggør målrettet optimering i systemet uden bekymring om at ødelægge andre komponenter i programmet. Artiklen konkluderer, at sandsynligheden for udbredelsen af SQLite er et resultat af dens platformoverskridende kode og filformat, kompakte og selvstændige bibliotek, omfattende testning og lave overhead.

Behovet for beskyttelse af brugerdata og for at opfinde nye former for at håndtere brugerdata er kun vokset løbende med udviklingen af internettet, især siden det første sociale media blev til. En ny metode til opbevaring og håndtering af brugerdata og indhold på sociale medier er opstået, men det er stadig en betydelig udfordring at finde rundt i. Det er op til udviklere at sørge for sikkerheden og beskyttelsen af brugerne samt hvordan deres data cirkuleres rundt i alle de forskellige medier, der i dag findes og stadig opfindes.

6.4 Delkonklusion

I arbejdet med at udvikle en platform for computerinteresserede er det væsentligt at anvende den viden, der er fremkommet gennem State of the Art.

Afsnit 6.1 har bidraget med en grundlæggende forståelse af sociale mediers rolle og indflydelse i dagligdagen. Det blev klart at sociale medier både skaber mulighed for nem og bred kommunikation, men samtidig også kan have negative konsekvenser såsom overfladisk interaktion og afhængighed. Denne indsigt er vigtig i forhold til at udvikle en platform, der understøtter fagligt fællesskab og fordybelse frem for hurtig og passiv brug. Afsnittet kommer også med konkrete eksempler på hvilke web-teknologier kan benyttes når man skal udvikle en velfungerende platform.

Gennem afsnit 6.2 blev det tydeligt, hvor afgørende brugerinddragelse og gennemsigtighed er for at opbygge et velfungerende online fællesskab. Erfaringer fra Reddit viser, at det ikke nødvendigvis er de mest højtråbende brugere, der repræsenterer flertallets ønsker. Det peger på behovet for at lytte bredt og kvalificeret til fællesskabet. Desuden giver indsigter om aktivt og passivt engagement nyttig viden til at designe funktioner, der skaber reel deltagelse. Derfor skal man som udvikler sørge for ordentlige brugertests og løbende tests for at sikre platformens fremgang.

Afsnit 6.3 satte fokus på privatliv, datasikkerhed og datalagring, hvor vi blev introduceret til en ny tilgang baseret på tillid og sociale normer frem for ren kontrol. Dette perspektiv er særligt relevant, da brugerne på en teknisk orienteret platform typisk har høje forventninger til gennemsigtighed og databeskyttelse. Samtidigt skal en velfungerende socialt medie have en robust sikkerhed for at beskytte brugernes data gennem ordentlig planlægning af API endpoints samt sikker data lagring.

Sammenfattende viser afsnittet, at en succesfuld platform bør bygge på en afbalanceret kombination af brugervenlighed, fællesskab og datasikkerhed. Det handler ikke blot om funktionalitet, men i høj grad også om at skabe et digitalt miljø, hvor brugerne føler sig engagerede, inddragede og trygge.

7 Designløsning

I dette afsnit beskrives den viden gruppen har opnået fra litteratursøgningen samt hvordan den viden kan benyttes til udviklingen af projektet, hvilke krav systemet skal opsættes med ved at lave undersøgelser i feltet med user stories, diagrammer og kravspecifikationer, samt implementering udfra disse krav og til sidst beskrivelse af løsningen.

7.1 Hvad har vi lært fra State of Art?

Litteratursøgningen har givet en bredere forståelse af gruppens valgte emne. For at kunne designe og udvikle en velfungerende platform skal man sørge for at kende sin målgruppe, hvilke teknologier man gerne vil benytte, samt hvordan man kan sikre brugernes sikkerhed.

For at kunne bygge et socialt medie fra bunden har man behov for at kende sin målgruppe, ved at undersøge hvordan brugere navigerer i de forskellige eksisterende platforme. Der er mange forskellige måder på at analysere og indsamle data fra brugere, både fra udviklernes side og fra virksomheder. Alt afhængigt at hvilke intentioner man har med undersøgelser, kan man lære en masse om hvordan brugerne færdes online. I afsnittet, som omhandler Reddit, har man lavet mange fejl og rettet disse til, ved at lytte til brugerne. En enkelt fejl kan koste mange ressourcer og et massetab af brugerne, og man skal arbejde på højtryk for at genvinde brugernes tillid. Samtidigt skal man huske at beskytte brugerne mod den altid voksende cyber trussel fra selve platformen og fra brugerne. Et hul i sikkerheden kan betyde stort tab af brugernes fortrolige oplysninger, og kan føre til en større mistillid til platformen.

Det er derfor vigtigt at have gensidig respekt fra begge sider, fra selve udviklerne og fra brugerne. SMPM modellen foreslår at platformene skal være bygget på kommunikation, tillid og sociale normer. Som udvikler skal man huske at overveje hvilke former for sikkerhed platformen skal udvikles med, og gerne have en så gennemsigtig kommunikation som muligt med brugerne om deres vilkår og hvordan deres data håndteres, før brugeren overhovedet oprettes i systemet. Derfor skal man også sørge for at lære brugerne at kende, blandt andet hvordan de deler indlæg, hvilke de interagerer med osv. COBRA modellen beskriver brugernes forbrug af indhold, og hvordan man kan drage fordel af, at udnytte disse undersøgelser til at øge engagement fra brugerne, dermed øge aktiveteten i selve platformen. I projektet kunne det overvejes at anvende disse metoder til at måle hvor meget engegement forskelligt indhold genererer. Denne data kunne bruges til flere forskellige ting, for eksempel kunne det være at fremme indhold der får brugere til at engagere mere. I dette tilfælde skal det dog også være vigtigt at tage typen af engegement der er tale om i betragtning, da engegement blandt andet også kan stamme fra negative følelser. Det kan for eksempel handle om artikler med positive hensigter, men hvor kommentarerne kan virke negative i forhold til indholdet, eller enkelte personer som kommer med falske informationer. Det kan drage til at den enkelte bruger ikke vil have lyst til at interagere med artiklen, da disse negative kommentarer kan få artiklen til at fremgå misvisende eller ukorrekt. Hvordan kan man som udvikler sørge for at fremme den gode tone fra brugerne, og sørge for at holde indholdet så troværdig og rent som muligt?

Samtidigt findes der et hav af falske informationer online, og brugere, som ikke har de bedste intentioner for andre brugere eller de platforme de bruger. I dette projekt vil vi gerne danne et sikkert sted for mennesker med interesse i teknologi. Ved at kigge på andre platforme med lignende strukturer, såsom Reddit, har gruppen dannet et større overblik over hvordan

projektet kan udformes. Ved at oprette moderatorer, som frivilligt kan vedligeholde deres egne forum, kan det give platformen mere troværdighed, da man vil kunne holde styr på hvilke opslag som kommer frem, og man kan fjerne uønskede brugere eller opslag. Det vil give mere autonomi til brugerne, som forhåbentlig vil give større udbytte i forhold til at holde platformen kørende og holde andre uønskede informationer ude. Det vil også være nemmere at holde dialog med brugerne og moderatorerne, og løbende opdatere teknologien i forhold til brugernes feedback.

Derfor er det også vigtigt at følge med i udviklingerne af de mange teknologier der findes, da disse opdateres kontinuerligt. Ved at arbejde med teknologier, som er i konstant udvikling, kan man aktivt mindske risikoerne ved cyber attacks og bedre beskytte brugernes sikkerhed. Som nævnt i afsnittet om API's har man længe døjet med hvordan data håndteres, da dette form for data arkivering er forholdsvis nyt, og man i dag stadig arbejder med at forbedre hvordan brugernes privatliv kan beskyttes online. Datalagring er også en vigtig del af udviklingen af projektet, og valget af hvilket teknologi er derfor essentiel. Afsnittet forklarer fordele og ulemper ved SQLite. I vores projekt har vi valgt at benytte SQLite for dens evne til at håndtere større arbejdsbyrder og skalerbarhed samt den nemme integrering til andre programmeringssprog. Vi har også kigget på de forskellige TypeScript værktøjer, hvilke biblioteker der findes, og hvilke former for databaser, som vil passe til den mængde som vores prototype kommer til at fylde. I projektet har vi fokuseret på at vælge programmeringssprog som er cross-platform kompatible, altså de kan nemt kompileres for både amd64, og for arm64 (se Afsnit 8.4). Udviklingen af frontenden skal passe med hvordan backenden og databasen er struktureret. Litteratursøgningen har givet en masse stof til eftertanke, og disse tager vi videre med i projektet.

7.2 Systembeskrivelse

Gnuf skal være en social medie platform, som primært er designet til brugere som har interesse i programmering, de nyeste teknologier indenfor tech-verdenen og andre tech-relaterede emner.

Platformen skal give brugere mulighed for at oprette personlige profiler, som være i stand tit at logge ind og ud af pågældende profil. Alt brugerens aktivitet skal knyttes til denne bruger. Hver brugeres profil skal kunne ses via en brugerprofil side, hvor deres brugernavn, og andre brugeroplysninger skal kunne ses. Desuden skal alle brugerens opslag også kunne ses på siden. Hvad angår dette, skal brugeren også være i stand til at oprette opslag, med en titel, hovedtekst og valgfrit billede. Derudover skal man også kunne interagere med eksisterende opslag i form af at like/dislike dem og at kommentere på dem. Opslagene bliver delt på specifikke fora kaldet "communities" i Gnuf. De kan oprettes, redigeres og slettes af sidens administratorer. Disse communities vil være opdelt baserede på forskellige emner, blandt andet programmering, tech nyheder og studie relatere ting. Hvert community har sin egen side, hvor der udelukkende bliver vist opslag der er oprettet på pågælende community. På den måde kan brugere nemt finde opslag som har noget at gøre med det de er interesseret i. Brugere kan også være medlem af communities, hvilket vil have indflydelse på hvilke opslag de ser på deres forside. Forsiden består af opslag fra de communities som brugeren er en del af i kronologisk rækkefølge startende fra det nyeste opslag. Hjemmesiden skal være fungerende i forhold til udviklingen af frontent, backend og database. Al data skal lagres sikkert for at sørge for brugernes datasikkerhed.

Grundlaget for platformen er at skabe engagerende og brugercentrerede oplevelser, og derfor skal systemet kunne håndtere store datamængder, som brugerne genererer.

7.3 User stories

For at danne et bedre overblik over brugeroplevelsen har vi nedskrevet nogle vigtige funktioner set fra brugerens perspektiv, både som normal bruger og som moderator. Formålet med dette er, at formulere hvordan en funktion i platformen kan give værdi til brugeren.

- Som bruger vil jeg gerne kunne oprette en profil eller logge ind på min eksisterende profil
- Som en bruger vil jeg gerne kunne interagere med andre ligesindede og udforske forskellige programmeringssamfunde online
- Som en bruger vil jeg gerne poste om mine interesser og komme med mine egne input til andres posts om programmering og de nyeste teknologier
- Som en bruger vil jeg gerne have muligheden for at tilføje et billede til min post.
- Som en administrator vil jeg være i stand til at administrere communities

Listen giver os et bedre overblik over projektets slutmål ved at benytte ikke-tekniske termer fra brugerens perspektiv til den videre udvikling af designløsningen.

7.4 Kravspecifikationer

De funktionelle og ikke-funktionelle krav benyttes til at identificere systemets egenskaber og funktioner. Det er en samling af krav, som vi har udviklet på baggrund af State of Art og user stories. De funktionelle krav beskriver, hvad systemet skal kunne fra brugerens perspektiv, mens de ikke-funktionelle krav fokuserer på, hvilke egenskaber systemet skal have for at kunne understøtte de funktionelle krav. I Tabel 1 og Tabel 2 beskrives hvilke krav systemet skal implementeres med, og indeholder en oversigt over systemets funktionelle og ikke-funktionelle krav, herunder kravnummer, kravtype, beskrivelse, kilde og prioritet.

Funktionelle krav

Krav nr.	Beskrivelse	Prioritering & Kommentar)
1	Brugerne skal kunne oprette en konto med e-mail	Must, fra user story
2	Brugerne skal kunne logge in med brugernavn og adgangskode	Must, fra user story
3	Brugeren skal kunne interagere i Communities	Should, fra user story
4	Brugeren skal kunne lave opslag med tekst og billede	Must, fra user story
5	Brugeren skal kunne like, dislike og kommentere på opslag	Must,fra user story
6	Brugeren skal kunne have en liste over de Communities de følger/er en del af	Could, fra user story
7	Startskærmen skal vise et udforsk feed over populære opslag	Could, fra state of art
8	Liste over brugerens Communities på deres profil	Should, fra user story
9	Brugeren skal kunne tilføje billeder til deres posts	Should, fra user story
10	Al data skal kunne lagres	Must, fra systembeskrivelse
11	Admins skal kunne oprette, redigere og slette communities	Should, fra user story
12	Brugeren skal kunne åbne hjemmesiden uden besvær.	Must, fra user story

Tabel 1: Funktionelle krav

Ikke funktionelle krav

Krav nr.	Krav	Kommentar
1	Systemet skal kunne håndtere mindst 40 brugere uden at blive langsom	fra krav 10
2	Systemet skal kunne loade billeder inden for 5 sekunder	fra krav 9
3	Systemet skal tage imod billeder med typerne filformat JPEG, PNG og GIF	fra krav 9
4	Adgangskoder skal lageres sikkert med hashing	fra krav 1 og 2
5	Der skal sikres at kun brugere med admin rettigheder kan slette communities	fra krav 11
6	Websiden skal kunne navigeres uden unødigt besvær af de fleste utrænnede brugere	fra krav 1, 2, 4, 5, 12
7	Hjemmesiden skal kunne åbnes, inden for rimelig tidsgrænse (10 sek)	fra krav 12
8	Systemet skal tage imod brugernavne på maks 100 karekter	krav 1 og 2
9	Backend skal skrive i C#	Projekt krav
10	Systemet skal inkorporere complex data managment	Projekt krav

Tabel 2: Ikke-funktionelle krav

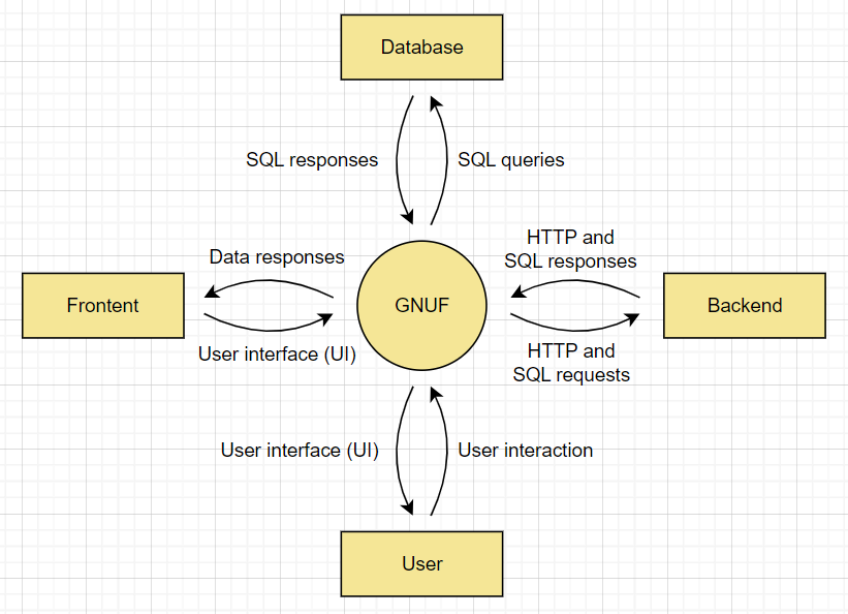
De overstående funktionelle og ikke-funktionelle specifikationerne kommer fra systembeskrivelse, user stories og selve kravende for dette semester projekt. De bruges som reference punkter løbende under udviklings processen. Det giver os en klar vison over de krav der er stillet for systemet, så vi bedre kan forme vores udviklings process ud fra priortering.

7.5 Diagrammer

Under projektet har vi udarbejdet flere diagrammer for at få et bedre overblik over systemets struktur og funktioner. Diagrammerne har hjulpet os med at visualisere, hvordan forskellige dele af systemet hænger sammen, og har gjort det nemmere at planlægge og fordele arbejdet i gruppen. Diagrammerne har hjulpet med at danne overblik over hvordan systemet er opsat, brugernes handlinger og hvordan backend og frontend snakker sammen. Derudover har de spillet en vigtig rolle i den agile arbejdsproces, hvor vi løbende har tilpasset og justeret vores løsning.

Gnuf context diagram

Figur 3 viser et context diagram. Diagrammet viser de overordnet terminator i systemet. Dette giver et indblik på, hvordan de forskellige dele i systemet interageres med.



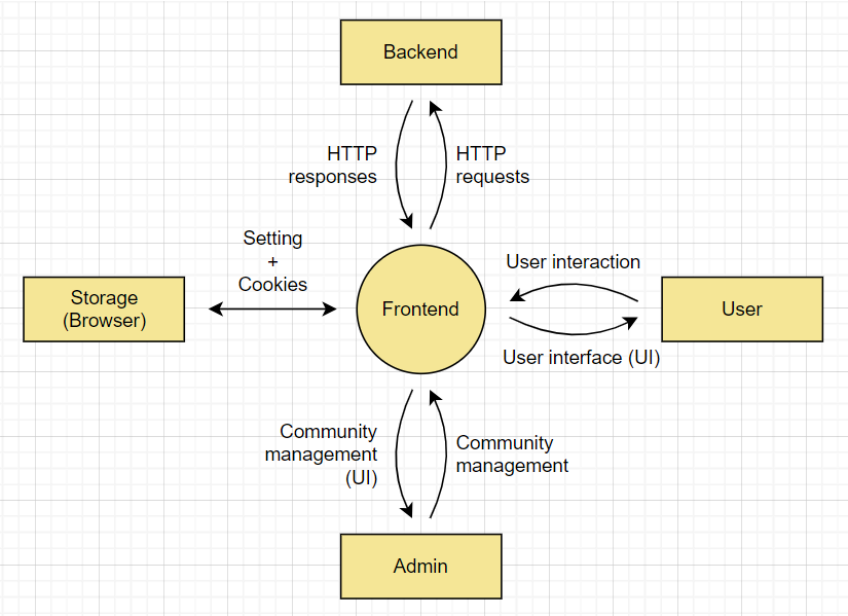
Figur 3: Overall context diagram

Terminatorer	Beskrivelse	Kommentar
User	Bruger platformen til at oprette, læse og interagere med indhold	• Har adgang til UI • Primær målgruppe: teknologientusiaster • Skaber og forbruger indhold
Frontend	Sender UI og modtager responses med data eller status	• Giver bruger adgang til systemfunktioner • Modtager input og returnerer relevant data
Backend	Håndterer data og funktionalitet via HTTP-anmodninger	• Modtager og svarer på forespørgsler • Behandler login, opslag og interaktioner • Sikrer datastruktur og sikkerhed
Database	Gemmer og leverer data, som backend anmoder om via SQL	• modtager data • gemmer data • organisere data

Tabel 3: Overall context table

Frontend context diagram

Figur 4 viser, hvordan de vigtigste aktører interagerer med GNUF's frontend. Frontenden fungerer som bindeled mellem brugerfladen, backend, browserens lagring og admin-funktioner. Tabellen nedeunder giver et kort overblik over hver aktørs rolle og relation til systemet.



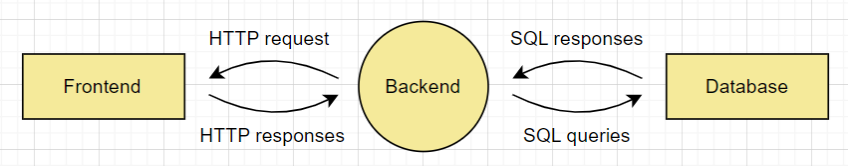
Figur 4: Frontend context diagram

Terminatorer	Beskrivelse	Kommentar
User	Bruger platformen til at oprette, læse og interagere med indhold	• Har adgang til UI • Primær målgruppe: teknologientusiaster • Skaber og forbruger indhold
Admin	Administrerer communities	• Vil kunne skabe communitites • Vil kunne redigere communities • Vil kunne slette communities
Backend	Håndterer data og funktionalitet via HTTP-anmodninger	• Modtager og svarer på forespørgsler • Behandler login, opslag og interaktioner • Sikrer datastruktur og sikkerhed
Storage (Browser)	Gemme indhold, lokal lagring af cookies og indstillinger	• Vil gemme data fra brugerne • Vil slette data ved brugersletning • Bidrager til en mere flydende brugeroplevelse

Tabel 4: Frontend context table

Backend context diagram

Figur 5 viser de vigtigste komponenter, som GNUF's backend kommunikerer med. Backend fungerer som mellemled mellem frontend og databasen, og har ansvaret for behandling af forespørgsler, validering og datastruktur. Tabellen giver et overblik over systemets interne data og ansvar.



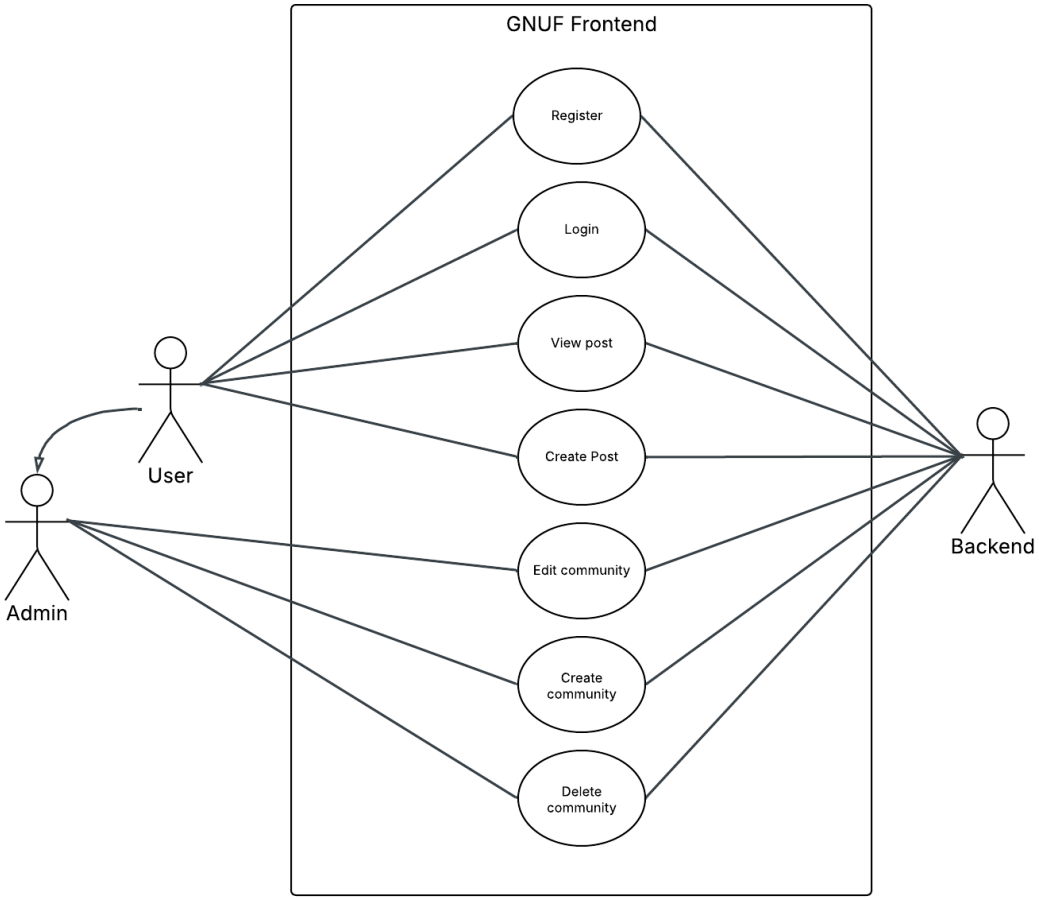
Figur 5: Backend context diagram

Terminatorer	Beskrivelse	Kommentar
Frontend	Sender HTTP-requests til backend og modtager responses med data eller status	• Giver brugerne adgang til systemfunktioner • Backend modtager input og returnerer relevant data
Database	Gemmer og leverer data, som backend anmoder om via SQL	• Håndterer brugerdata, posts, kommentarer og community-information • Backend laver forespørgsler og modtager struktureret svar • SQL bruges som forespørgselssprog i systemet

Tabel 5: Backend context table

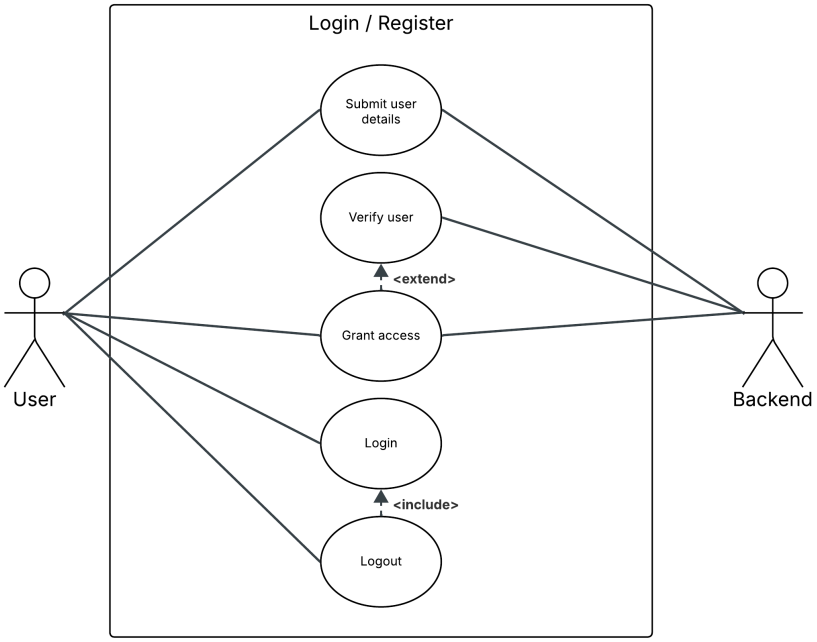
Use case diagrammer

De følgende diagrammer illustrerer et sæt af use cases for projektets system ved brug af aktører og deres relationer. Der er blandt andet en use case for frontend delen af udviklingen med aktørerne "Admin", "User" og "Backend", og de andre use cases beskriver hvordan relationerne fungerer når brugeren udfører andre opgaver i systemet.



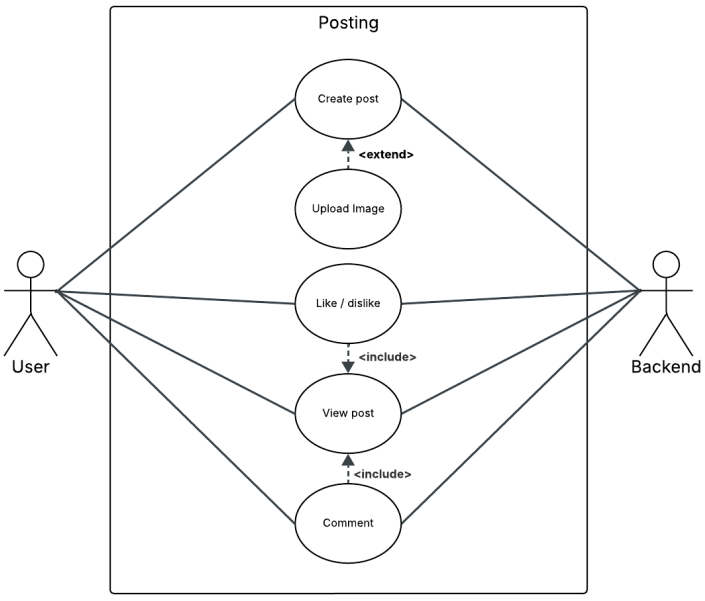
Figur 6: Use case diagram for Frontend

Som ses på den overordnede use case diagram på Figur 6 er der 3 aktører, *User*, *Admin* og *Backend*. Systemets grænse indeholder frontend systemet, og her beskrives hvad hvert aktør kan udføre og hvordan de integrerer med hinanden. *User* er i stand til at registrere sig i systemet samt logge ind, og her ses det på *<include>* fra *Login* til *Register*, at man ikke kan logge ind uden først at have registreret sig. Brugeren kan også lave og se indlæg ved *Create post* og *View* post. Aktøren *Admin* kan udføre alle handlinger som *User*, og kan lave forummer, opdatere og slette disse ved *Create community*, *Delete community* og *Edit community*. Alle handlinger berører aktøren *Backend*, da disse data fra frontend skal lagres i backenden for at fungere.



Figur 7: Use case diagram for Login/Register

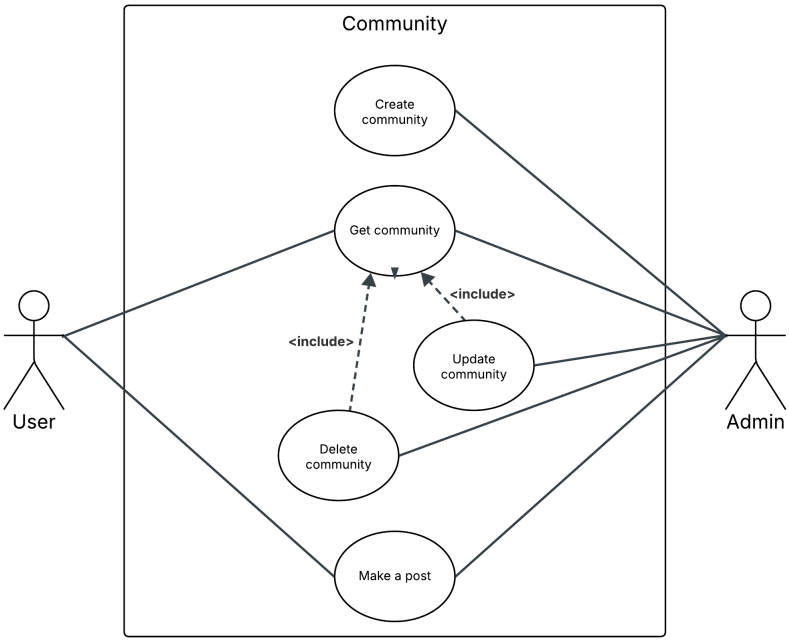
Figur 7 beskriver hvordan en bruger kan oprettes og logges ind i systemet, og inkluderer aktørerne *User* og *Backend*. Brugeren indsender registreringsinformationerne, som modtages af *Backend*, hvorefter de verificeres. Efterfølgende får brugeren adgang til systemet ved *Grant access*. Brugeren kan efter reginstrering selv logge ind og logge ud af systemet.



Figur 8: Use case diagram for Posting

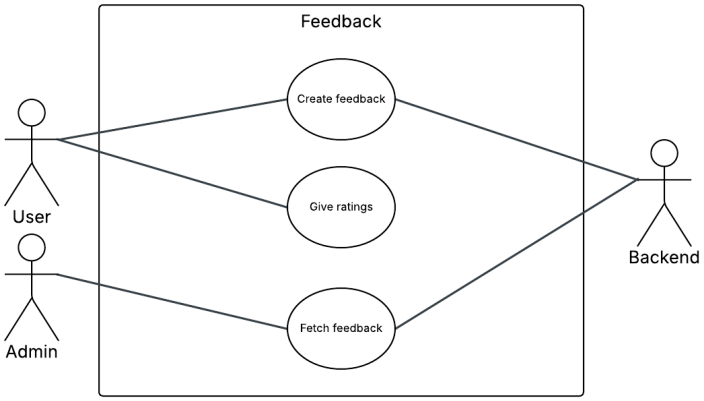
Figur 8 beskriver hvordan systemet håndtere oplag, og her beskrives aktørerne som *User* og *Backend*. *User* er i stand til at lave et indlæg, *Create post*, og ved at lave et indlæg tilbydes det at inkludere *Upload image*, dvs. brugeren kan vælge at inkludere et billede til indlægget. Brugeren kan også kigge på egen eller andres indlæg, *View post*, hvor brugeren kan trykke på *Like/ dislike* eller *Comment*, og som set på figuren, så kan man kun interagere med et indlæg, hvis

den allerede findes, derfor kan man kun like/dislike og kommentere hvis indlægget eksisterer. Alle handlinger gemmes i aktøren *Backend*



Figur 9: Use case diagram for Community

Figur 9 beskriver hvordan *Community* fungerer, og har aktørerne *User* og *Admin*. Brugeren kan blive del af en community ved *Get community* og kan lave indlæg ved *Make a post*. Alle handlinger håndteres af *Admin*, og inkluderer *Create community*, *Update community*, og *Delete community* udover *Get community* og *Make a post*.

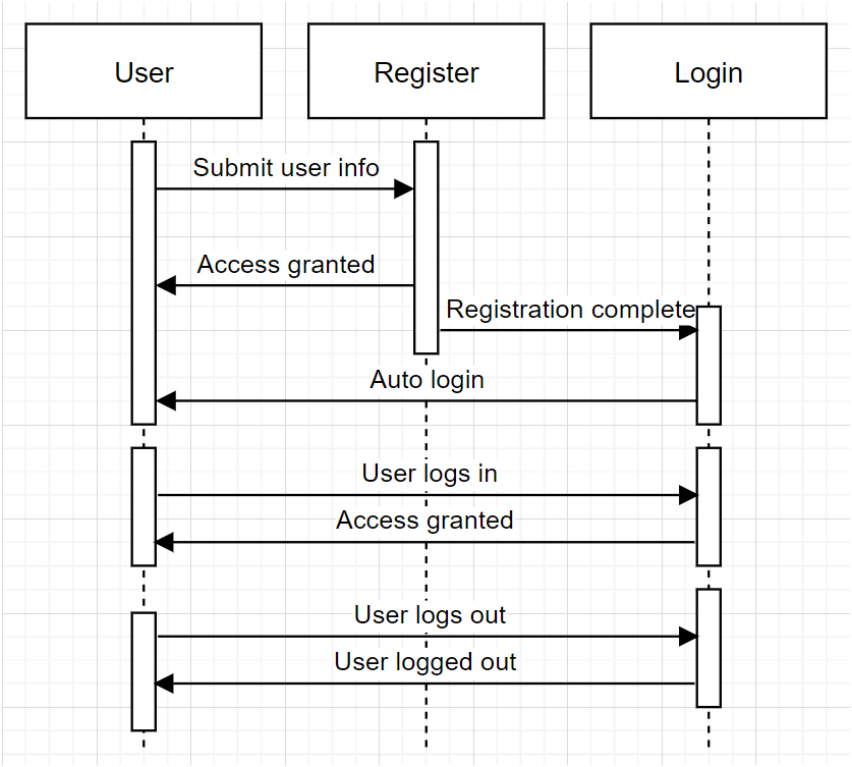


Figur 10: Use case diagram for Feedback

Figur 10 beskriver hvordan feedback håndteres, og indeholder aktørerne *User*, *Admin* og *Backend*. Brugeren kan skrive feedback og kan bedømme ved *Create feedback* og *Give rating*. Admins kan få adgang til disse med direkte database opslag

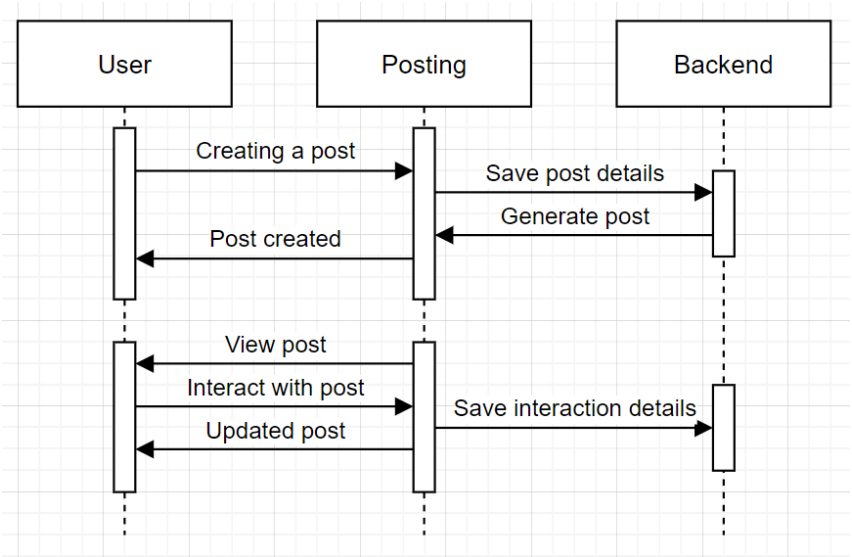
Sekvensdiagrammer

De følgende diagrammer viser procesinteraktionerne arrangeret i tidssekvens. Sekvensdiagrammerne er udarbejdet på baggrund af use case-diagrammerne.



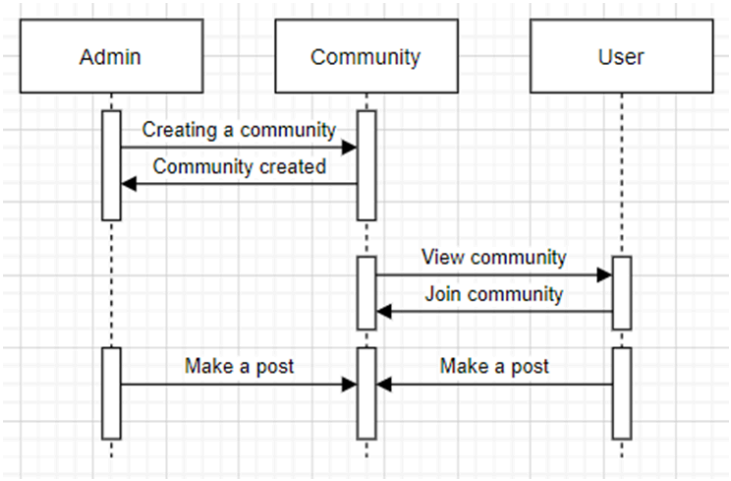
Figur 11: Sekvens diagram for Login / Register

Figur 11 viser hvordan brugeren registreres i systemet, ved at indsende sine informationer til systemet. Herefter logges brugeren ind automatisk, og kan påbegynde at skrive indlæg. Brugeren kan herefter vælge at logge ud af systemet, og kan på et senere tidspunkt logge ind i systemet igen.



Figur 12: Sekvens diagram for Posting

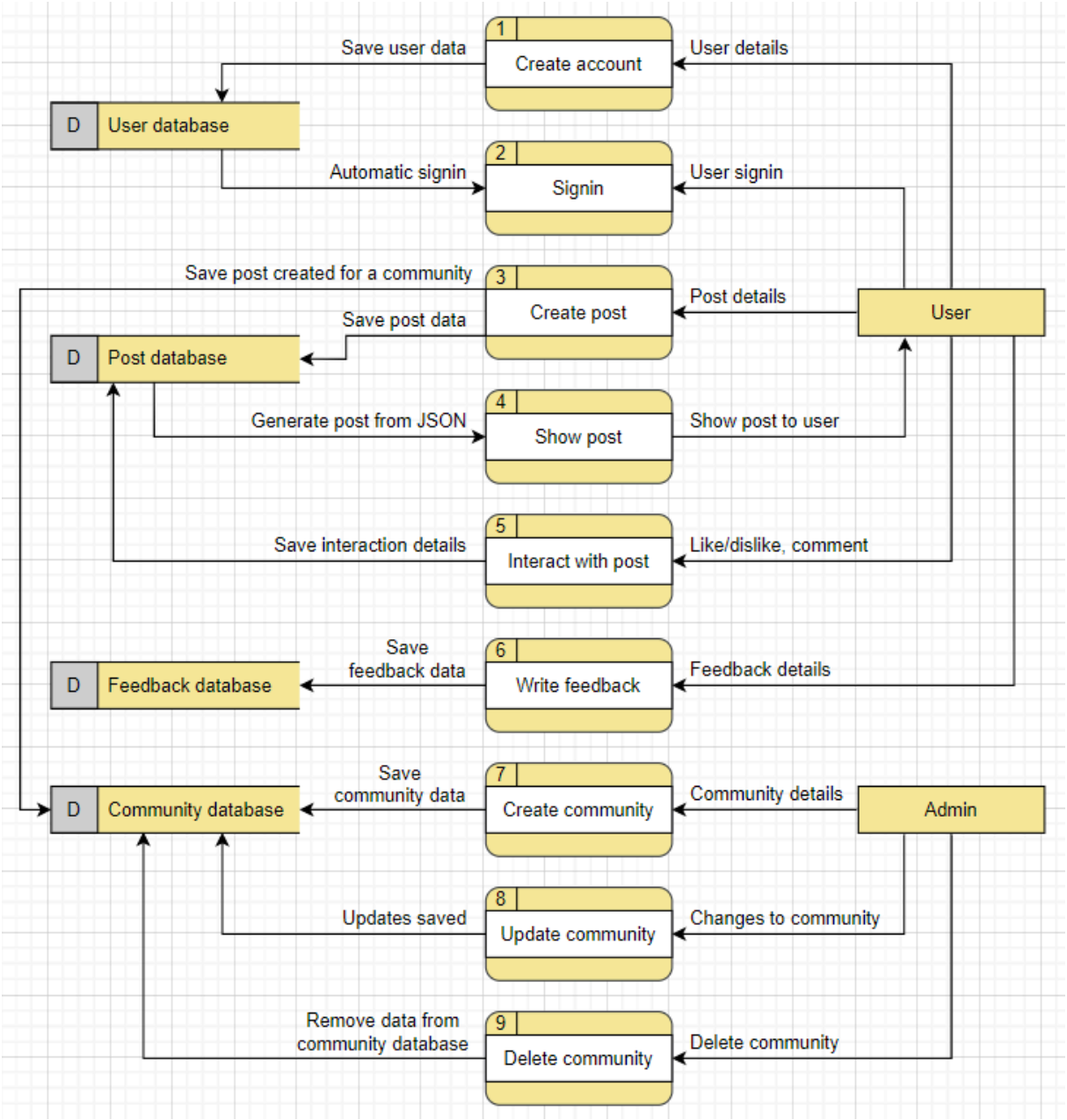
Figur 12 beskriver hvordan brugeren laver indlæg i platformen, hvor første trin er at lave indlægget, som herefter gemmes i systemets backend, som genererer opslaget, og brugeren får bekræftelse at opslaget er skabt. Brugeren kan også se andres opslag og interagere med disse ved at like/dislike og kommentere, og disse data gemmes i systemets backend, som herefter opdateres og vises til brugeren.



Figur 13: Sekvens diagram for Community

Data flow diagram

Data flow diagrammet kortlægger informationsstrømmen for processen gennem systemet, fra brugerens profil oprettelse i systemet samt oprettelsen af indlæg, communities og feedback.



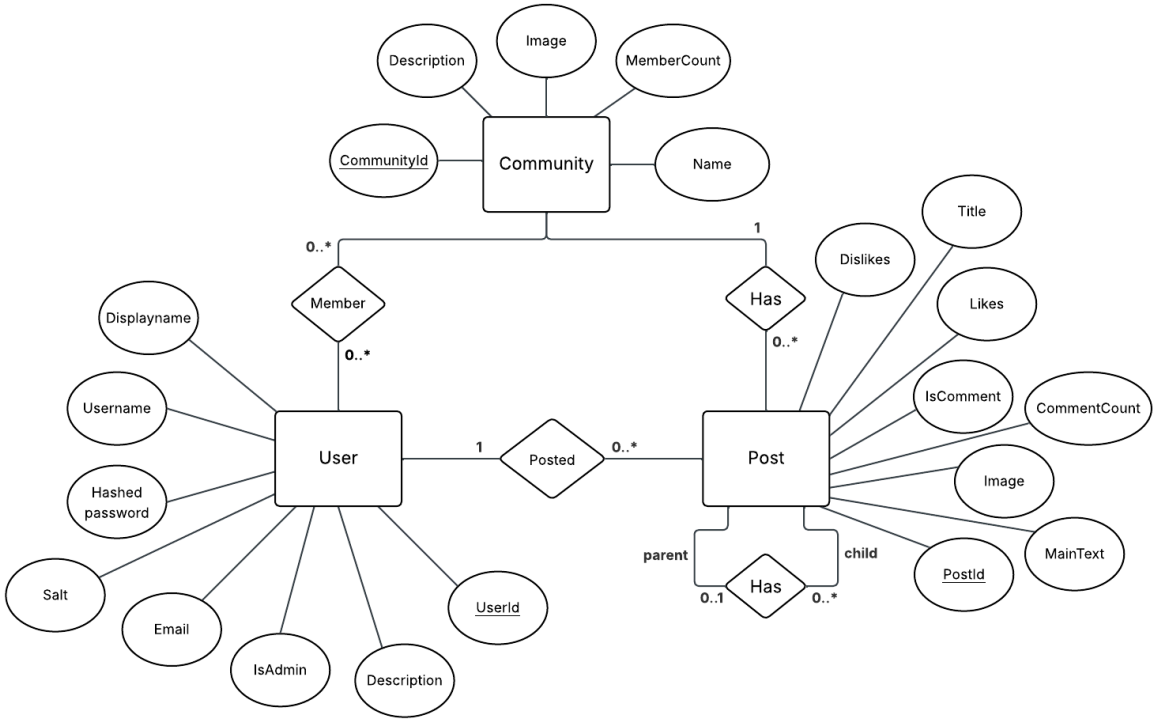
Figur 14: Data flow diagram

Figur 14 viser enhederne *User* og *Admin*, datalagrene *User database*, *Feedback database* og *Community database* samt 9 processer gennem systemet. Ved processen *Create account* putter brugeren sine informationsdetajler, og disse lagres i *User database*. Ved *Create post* laver brugeren et indlæg, og dette lagres i *Post database*, som genererer JSON som herefter vises på platformen. Brugeren interagerer med indlægget ved enten at *Like/dislike* eller kommentere på indlægget, og disse data lagres igen i *Post database*. Endvidere kan brugeren vælge at skrive et feedback ved processen *Write feedback* og dette lagres i *Feedback database*, som kan tilgås af *Admin* ved at hente direkte fra databasen, som forklaret på Use case diagrammet på Figur 10.

Samtidigt kan *Admin* skabe, redigere og slette et community, og disse informationer gemmes i *Community database*

Entity Relationship diagram

ER diagrammet beskriver hvordan hvert enhed relaterer til hinanden i systemet, i det her tilfælde *User*, *Community* og *Post* og hvilke attributter hvert enhed indeholder.



Figur 15: ER diagram

Enheden *User* indeholder blandt andet *Displayname* og *Username*, *Email* og *Hashed password*, og *UserId* som Key Attribute, som betyder at ID'et er unikt for brugeren. Brugeren kan lave ét indlæg ad gangen, som ses på 1-tallet fra *User* til *Posted*.

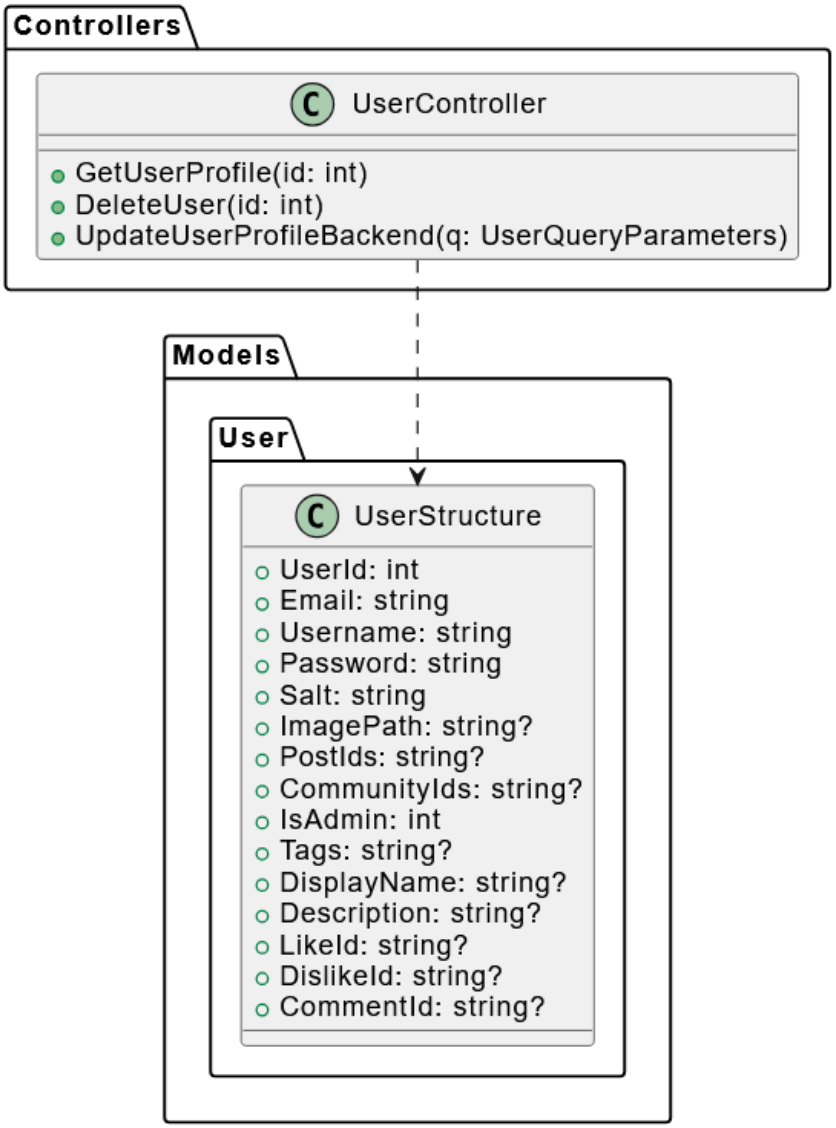
Brugeren kan lave flere indlæg, som ses på *0..** fra *Posted* til *Post* og et indlæg, *Post*, indeholder blandt andet *Likes* og *Dislikes*, *IsComment* for hvorvidt indlægget indeholder kommentarer, *Image* for billede inkluderet i indlægget, og *PostId* som Key Attribute. Under *Post* ses forholdet mellem indlæg og kommentarer afhængig af om *Post* er *parent* eller *child* til et andet indlæg. I GNUF laves der en "ny" post når en bruger kommenterer på indlægget, og forbinder til den originale indlæg. I diagrammet ses det, at en *Post* kan have én *parent* ved at kigge på tallen *0..1*, dvs. et indlæg, som ikke er den originale, skal linke til en enkelt *parent* indlæg. Samtidigt kan en *Post* have mange "children", hvor *child* kan være flere end én ved tallet *0..**, som betyder at mange brugere kan kommentere på den originale indlæg, og hver af disse kommentarer skaber et nyt indlæg, som ikke forbinder til andre kommentarer, men forbinder til den *Post* man kommentarer på.

En *Post* kan også være en del af en *Community*, ligesom en *User* kan være *Member* af en *Community*. *Community* enheden indeholder blandt andet *Name*, *MemberCount*, *Image***{3}** og *Description*, og har *CommunityId* som Key Attribute. En *User* kan være medlem af mange

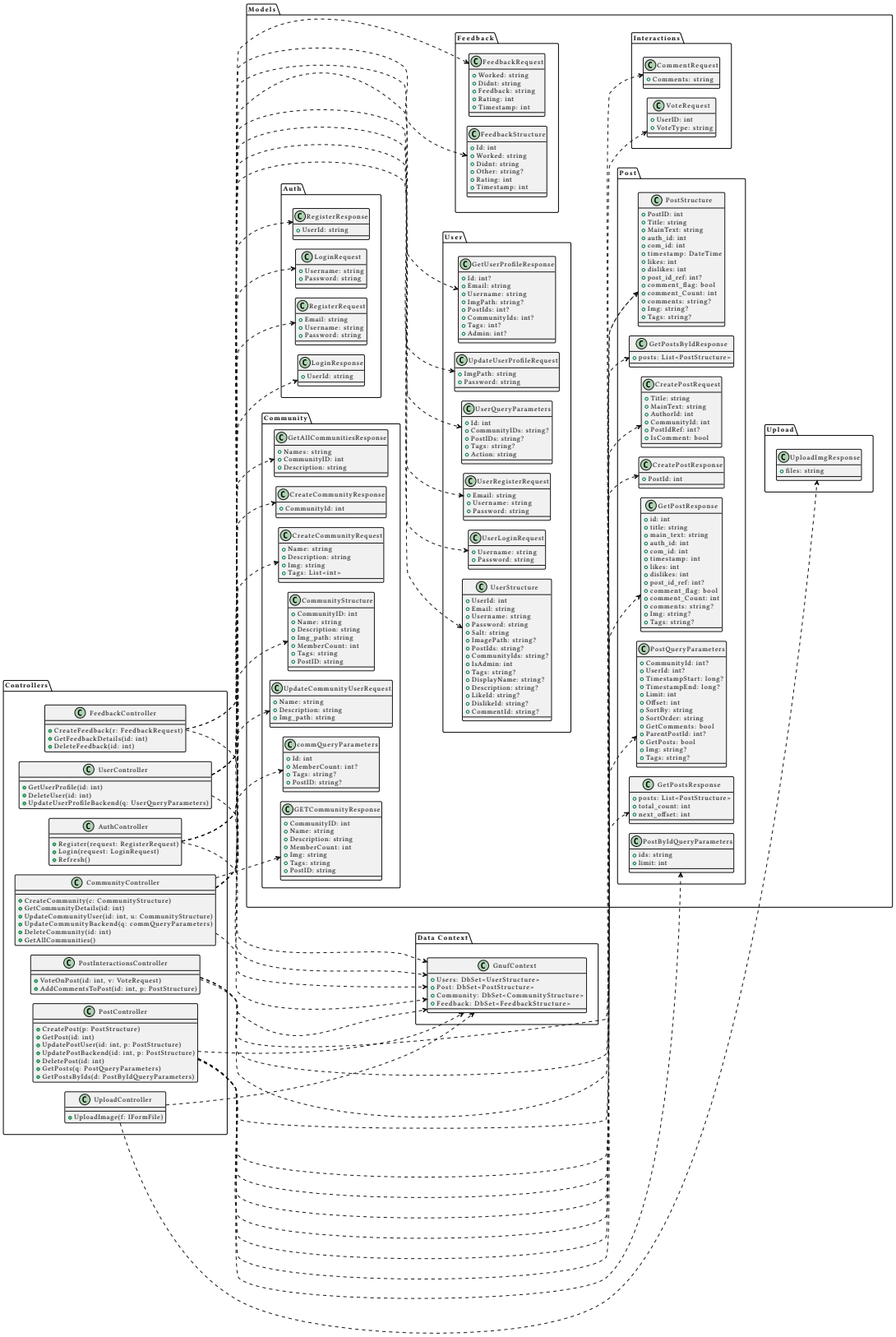
Communities, ligesom en *Community* kan have mange medlemmer. Derudover kan en *Community* indeholde en masse forskellige *Posts*, dog kun én af den samme *Post* ad gangen.

Klasse diagram

Klasse diagrammet beskriver strukturen af systemet, hvilke variabler klassen indeholder, og hvordan hvert klasse relaterer til hinanden. I diagrammet ses *Controllers*, *Models* og *Data Context*. Under *Controllers* findes for eksempel *UserController*, som indeholder variablerne *GetUserProfile*, *DeleteUser* og *UpdateUserProfileBackend*, som henter bruger data fra modellen *User* og reagerer på brugerinput og udfører interaktioner på datamodelobjekterne. Den er for eksempel forbundet til *UserStructure*, modellen som gemmer på brugerens data, blandt andet ved variablerne *UserId*, *Email*, *Username* osv. Disse klasser og deres variabler forklares mere dybdegående under afsnittet *Implementation*, under afsnittet ***Controllere***.



Figur 16: Klasse diagram for user



Figur 17: Klasse diagram

8 Implementation

Dette afsnit beskriver implementeringen af designløsningen med fokus på systemets tre hovedkomponenter: database, frontend og backend. Frontenden er udviklet i React, et open source JavaScript-bibliotek, databasen er lavet med SQLite, en fri objekt-relationel database server, og backenden er udviklet i C# og fungerer som en bindeled mellem brugergrænsefladen og databasen.

8.1 Frontend implementering

TypeScript er blevet anvendt til implementeringen af projektets frontend udvikling. TypeScript er et gratis open source programmeringssprog, som er designet til udvikling af store applikationer, som tilføjer statiske typer og transpilerer til Javascript. Den dynamiske fejlfinding og bedre læsbarhed af koden, samt noget erfaring med lignende sprog i gruppen, har gjort TypeScript til et oplagt valg for gruppen. Tanken bag gruppens projekt fra starten har været at udvikle en web-applikation med dynamisk indhold, og data som prioritet samt nem programmerings samarbejde.

TypeScript/JavaScript blev besluttet som det mest passende programmeringssprog til dette projekt da frontenden vil være en webapplikation med meget dynamiske indhold. Selvom det er muligt at udvikle dette i andre programmeringssprog, er JavaScript langt det mest anvendte. Browsere afvikler primært JavaScript, så hvis andre programmeringssprog anvendes skal det konverteres enten til JavaScript eller Web Assembly. Udover det er langt de fleste teknologier og værktøjer til webudvikling lavet til JavaScript, så ved brug af alternative sprog kan disse ikke anvendes. Derved vil udviklingen af front-enden yderligere kompliceres da alternative måder at opnå samme funktionalitet skal findes/udvikles.

React

Som tidligere nævnt vil web-applikationen, grundet den måde sociale medier grundlæggende fungerer på, indeholde meget dynamisk indhold. Med dynamisk indhold er det en stor fordel at anvende et bibliotek som er udviklet til formålet, da det kan være kompliceret at gøre med ren JavaScript. Disse biblioteker tilbyder desuden mange andre fordele for udviklingen. Det er her React kommer ind i billedet.

React er et JavaScript bibliotek brugt til UI udvikling. React bruger en komponentbaseret arkitektur, som udover at gøre dynamisk indhold nemmere at arbejde med, også kommer med mange fordele i forhold til udviklingen [18]. Da man kan opdele alle de forskellige dele af applikationens UI-elementer i genbruglige komponenter, bliver det meget nemmere at organisere koden, gør det mere læsbart og generelt bedre at arbejde med.

React tilbyder også funktionalitet for tilstandshåndtering, hvilket er endnu et vigtigt element i dynamiske web-applikationer **{1}**. Det gør det muligt at opdatere UI'et nemt og effektivt, så snart data ændres uden at skulle reloade hele siden. React har udover det også et stort økosystem med yderligere biblioteker og frameworks. Da vi også ønskede at anvende Next.js, et af disse frameworks som bygger på React, blev React et naturligt valg.

Next.js

Next.js fungerer som det underliggende React framework for applikationen. Det strømligner udviklingsprocessen og formindsker mængden af "boilerplate" kode[1] **{1}**, ved at automatisere mange opgaver som man ellers manuelt skulle implementere i React [19]. Blandt andet kan TypeScript og ***TailwindCSS*** automatisk sættes op af Next.js under opsætningen af projektet, hvilket er teknologier, der allerede var valgt til brug i projektet før Next.js. En af de væsentlige aspekter som Next.js håndterer er routing[2] og navigation. I et traditionelt React projekt vil man typisk skulle anvende et bibliotek som "React Router" som manuelt skal sættes op til at rutes til alle dele af applikationen. Next.js derimod, anvender en filbaseret tilgang hvor man bare kan oprette filer/mapper i `/app` mappen, og ud fra det bliver alle ruter automatisk genereret. Dette system vil blive forklaret mere i dybden i afsnittet ***Overordnet struktur***. Derudover tilbyder Next.js også adskillige optimeringer som forbedrer den færdige applikations ydeevne.

Det skal nævnes at Next.js er ment som et full-stack framework, og ikke blot et frontend framework som det bliver brugt i dette projekt. På trods af dette, fungerer det stadig helt fint selvom det udelukkende bruges til frontend delen, og man kan sagtens anvende det med en dedikeret back-end uden problemer. Man går blot glip af en række funktioner som server-side rendering af komponenter, og det er også en smule mere besværligt for frontend og backend at kommunikere med hinanden.

TailwindCSS

Til UI styling er TailwindCSS blevet anvendt, et CSS-framework der gør designprocessen enklere. I stedet for at have CSS filer med alle de forskellige CSS-klasser, styles der i stedet direkte i HTML-strukturen i form af klasser [20]. Komponenternes udseende defineres præcis der hvor komponenten anvendes, dermed gøre designprocessen mere strømlignet, da man ikke behøver konstant at navigere mellem HTML og CSS-filer. Udover det behøver man ikke konstant at finde på nye CSS-klasser og holde styr på hvilke der eksisterer. Det kan argumenteres at TailwindCSS har den bivirkning at man gentager styling mange gange i stedet for at genbruge CSS klasser. I realiteten er dette dog langt fra tilfældet. Især når der skal samarbejdes mellem flere personer, kan det blivet meget svært at holde styr på alle de forskellige klasser når ens applikation når en vis størrelse. Derved ender man ofte med at lave mange forskellige klasser i stedet for effektivt at genbruge dem. Når TailwindCSS kombineres med React reduceres dette problem da man med React også kan lave genbrugelige komponenter, og dermed formindske repetition i koden.

Shadcn

Shadcn er et komponentbibliotek som der i dette projekt er anvendt som et fundament for projektets komponenter og UI [21]. Shadcn tilbyder en stor samling af minimalt designede komponenter, som der nemt kan tilpasses ved brug af TailwindCSS. Dette forenkler udviklingen af applikationens UI, da der allerede er et grundlag for alle de forskellige komponenter, og det derfor ikke skal udvikles fra bunden af.

[1] Gentagne kodeafsnit der skal inkluderes i mange dele af et program med ringe eller ingen ændringer.
[2] Teknikken der dirigerer brugere til forskellige sider baseret på det URL de navigerer til

Andre biblioteker

Ydereligere biblioteker anvendes også til specifikke formål, såsom input validering og autentifikation:

- Zod
- Auth.js

Disse biblioteker introduceres i den kontekst de anvendes i senere i implementerings afsnittet.

Overordnet struktur

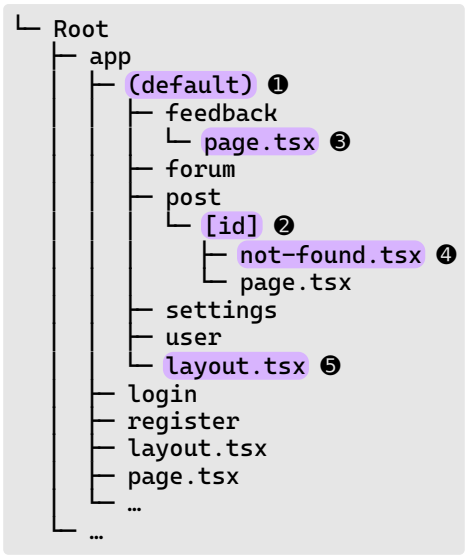
I dette afsnit gennemgås den overordnede struktur af frontend-applikationen, altså hvordan projektets filsystem er opbygget. Filstrukturens hovedformål er organisatorisk, og er bygget op for at være nem at vedligeholde og navigere. I visse tilfælde tjener det dog også et funktionelt formål, eksempelvis dannes applikationens routing direkte ud fra hvordan filsystemet er struktureret på. I Filstruktur 1 vises en overordnet visualisering af de vigtigste mapper i projektets rodmappestruktur. `/app` mappen håndterer applikationens sider og routing, `/components` indeholder de genbrugelige UI-komponenter, og i `/lib` findes logik for datahåndtering samt andre essentielle funktioner. De følgende underafsnit vil uddybe indholdet og formålet med disse mapper.

```
└─ Root
   ├─ app
   ├─ components
   ├─ lib
   ├─ node_modules
   └─ …
```

Filstruktur 1: Filstrukturen af applikationen

Sider og routing

Som nævnt tidligere anvender GNUF det indbyggede routing system tilbudt af Next.js. De forskellige ruter bliver generet baseret på opbygningen af `/app` mappen [22]. Hver mappe i `/app` repræsenterer et rute segment der svarer til et URL-segment. Bestemte filer i disse mapper definerer det UI, der bliver vist for hver rute. Et udsnit af `/app` mappen i GNUF's filsystem kan ses i Filstruktur 2. Bemærk at det kun er bestemte mapper som har deres filer inkluderet.



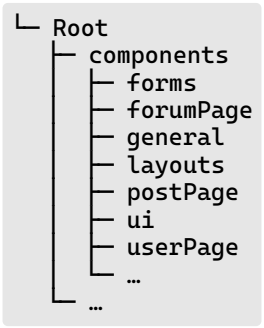
Filstruktur 2: Filstrukturen af `/app`

Øverst i `/app` befinder sig en speciel type mappe `(default)` ❶. Dette er et "rute gruppe", kendetegnet med parenteser der indkapsler navnet. Rute grupper bruges til at organisere ruter uden at have en effekt på URL-stien. I GNUF anvendes det til at separere den del af applikationen, der indeholder det sociale medie med resten af hjemmesiden (bl. a. login siden og "about us" siden). Dette tillader os at have et bestemt layout for alle ruter i denne gruppe. `[id]` ❷ er en anden speciel mappe, og repræsenterer et dynamisk segment. Det er kendetegnet med de kantede paranteser ("[]"), og kan indfange enhver værdi i dens URL-position. Denne værdi kan derefter bruges til fx. at hente specifik data baseret på denne værdi.

I `feedback` mappen ses der en `page.tsx` fil ❸. Dette er en speciel fil, der definerer det UI der som standard bliver vist til brugeren for den rute. Et andet eksempel på en speciel fil er `not-found.tsx` ❹, som definerer det UI som bliver vist, hvis siden ikke er blevet fundet. Dette er primært brugbart for dynamiske ruter, hvor der ikke nødvendigvis er noget data for en bestemt værdi. Til sidst er der også en `layout.tsx` ❺, som skaber et delt UI layout som anvendes af alle ruter inde for dens omfang. I dette tilfælde er det alle sider i `(default)` gruppen. Layoutet fungerer som en slags ydre skal, der indkapsler resten af koden.

Komponenter

GNUF gør i høj grad brug af genbrugelige komponenter som er en del af filosofien bag React. Disse komponenter findes i `/components` mappen. Strukturen på mappen kan ses i Filstruktur 3.



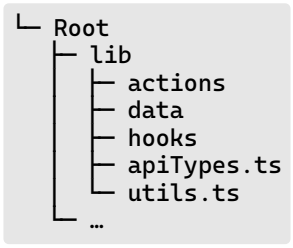
Filstruktur 3: Filstrukturen af `/components`

Komponenterne er organiseret i forskellige mapper baseret på hvad de bruges til. Fx. er alle formular (form) elementer i `form` mappen, komponenter som specifikt bruges i siden til opslag findes i `postPage`. `general` mappen indeholder komponenter som bruges flere steder, og `ui` indeholder fundamentale komponenter som bruges til at konstruere de resterende komponenter.

Det kan diskuteres om der i et større projekt burde være endnu mere organisering, hvor man bla. kunne separere side-specifikke mapper med generelle mapper. Dog af et projekt på denne størrelse fungerede det anvendte system fint.

Datahåndtering og andre funktioner

I `/lib` mappen findes den kode, der binder applikationens UI sammen med backend, samt diverse hjælpefunktioner. Strukturen kan ses i Filstruktur 4.



Filstruktur 4: Filstrukturen af `/lib`

Mappen `actions` indeholder kode der anvendes til at sende en HTTP POST-request til backenden, og som typisk får den til at udføre en form for handling. Dette kan bl. a. være at oprette et opslag, eller lave en ny bruger-profil. I `data` er der kode, der bruges når der bare skal hentes data fra backenden i form af en HTTP GET-request, og `hook` indeholder brugerdefinerede React hooks. Alt dette vil blive forklaret yderligere senere i rapporten.

Kernekoncepter

Applikationens funktionalitet bygger på en række tekniske elememter. Dette omfatter blandt andet håndtering af brugeradgang, datainsamlig via formularer og kommunikation med applikationens backend.

Autentifikation og autorisation

For et socialt medie som GNUF som afhænger af brugergenereret indhold, er brugerprofiler et centralt og nødvendigt element. Størstedelen af de planlagte funktioner afhænger af brugerprofiler, og derfor var det en at de ting, der var højest prioriteret i projektet. For at brugerprofiler fungerer, er applikationen nødt til at implementere et autentifikation og autorisation system. Dette giver mulighed for at verificere hvem hver bruger er og hvilket bruger-profil de har adgang til, samt at afgøre hvad de har tilladelse til at gøre.

GNUF gør brug af Auth.js, et bibliotek der anvendes til at implementere et komplet autentificeringssystem. Dette blev besluttet da det er yderst besværligt at implementere et autentificeringssystem fra bunden af. Grundet størrelsen af projektet og tidsbegrænsningen af GNUF var denne tid bedre brugt andre steder, som at implementere **{1}** kernefunktionalitet til applikationen. Udover det åbner et egenudviklet autentificeringssystem døren for utallige sikkerhedshuller, og vil højst sandsynligt også være et suboptimal system i forhold til Auth.js.

De følgende kode udsnit viser nøglekomponenterne i autentificeringssystemet i forhold til sikkerhed og funktionalitet. I Kode Blok 1 ses den fundamentale konfiguration af Auth.js, hvor der opsættes en “credentials provider”. Her anvendes et traditionelt brugernavn/password flow, hvor en bruger kan logge ind på den konto et brugernavn knyttet, til hvis de giver den korrekte brugernavn/password kombination.

```ts
export const { handlers, signIn, signOut } = NextAuth({ ❶
  providers: [
    CredentialsProvider({
      credentials: {
        username: {},
        password: {},
      },
      authorize: async (credentials) => { ❷
        const requestData: UserLoginRequest = {
          username: credentials.username as string,
          password: credentials.password as string,
        };

        const response = await fetch(`${process.env.API_URL}/api/auth/login`, { ❸
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(requestData),
          credentials: "include",
        });

        if (!response.ok) return null;

…                                   { cookie handling… }

        const user = await response.json();
        return user;
      },
    }),
  ],
…                                           { … }
});
```

Kode Blok 1: Udbyder-konfiguration og login logik (`/auth.ts` linje 69-112)

For at anvende Auth.js skal det først initialiseres. Dette gøres med funktionen `NextAuth`, som modtager forskellige konfigurationer som parameter ❶. Den kan returnere en stor mængde metoder, hvor de fleste dog er irrelevante for GNUF. Blandt de relevante metoder er `handlers`, som i dette projekt bruges til at expose de API endpoints som Auth.js bruger til at håndtere sessioner. `signIn` og `signOut` er funktioner som kan bruges i applikationen til at forsøge at logge ind og ud.

Under `CredentialsProvider` implementeres en tilpasset `authorize` funktion ❷, som fortæller Auth.js hvordan den skal håndtere de loginoplysninger, systemet modtager fra brugeren når de forsøger at logge ind. Her sendes en request med loginoplysningerne til det backend API endpoint som håndterer login ❸. Der kan læses mere om hvordan dette endpoint fungerer i ***AuthController (4.1)*** under ***User Login***. En vigtig indstilling er `credentials: "include"`, som sørger for at cookies bliver inkluderet, hvilket er nødvendigt for backenden så den kan sætte cookies i dens response.

For session-håndtering anvendes en JWT-baseret session. En JWT (JSON Web Token) er en selvstændig token, der kan anvendes til sikkert at overføre information. JTW'er indeholder brugerinformation som bruges til at opretholde sessionen, samt en signatur som beviser at den er autentisk og ikke er blevet ændret. Der kan læses mere om hvordan JWT'erne genereres i ***Services*** under ***Tokenhåndtering***. I GNUF er denne strategi valgt da der ikke er behov for at databasen håndterer sessioner, hvilket forenkler processen markant [23]. I Kode Blok 2 kan der ses hvordan JWT og sessioner håndteres i Auth.js.

```ts
export const { handlers, signIn, signOut, auth } = NextAuth({
…                                                  { … }
  callbacks: {
    async jwt({ token, user }) { ❶
      if (user) { ❷
        const customUser = user as CustomUser;
        token.id = customUser.userId;
        token.accessToken = customUser.accessToken;
        token.expiresAt = Date.now() + tokenExpiration;
…                                      { other properties… }
        return token;
      }

      if (token.expiresAt && typeof token.expiresAt === "number") { ❸
        if (Date.now() < token.expiresAt) return token;
        return refreshAccessToken(token);
      }
      return token;
    },

    async session({ session, token }) { ❹
      if (token && session.user) {
        session.user.id = token.id as string;
        session.accessToken = token.accessToken as string;
…                                      { other properties… }
        if (token.error) session.error = token.error as string;
      }
      return session;
    },
…                                                  { … }
});
```

Kode Blok 2: JWT og session (`/auth.ts` , linje 118-167)

`jwt` ❶ er en callback funktion som kaldes hver gang en JWT generes (f. eks. under login) eller opdateres (f. eks. når en session tilgås af Auth.js). Her inkluderes et `token` objekt som parameter, som er den JWT som Auth.js administrer. Når en bruger autentificeres som følge af at logge ind eller registrere en konto, inkluderes også det `user` objekt der retuneres af `authorize` funktionen (se Kode Blok 1). I dette tilfælde opsættes `token` objektet baseret på indholdet af `user` . Ved efterfølgende kald til `jwt` , tjekkes der om den nuværende token stadig er gyldig ❸. Hvis den er, retuneres `token` , men ellers vil systemet forsøge at forny den

med `refreshAccessToken` funktionen.
`session` ❹ er en anden callback funktion som kaldes når en session tjekkes (f. eks. data i den tilgås). Som parameter inkluderes der et objekt der også hedder `session`, som er det objekt Auth.js returnerer når man tilgår en session. I GNUF's filfælde indkluderer den præcis det samme information som `token` objektet.

Auth.js har også en callback funktion til autorisationslogik som sørger for det sociale medie kun kan tilgås hvis man er logget ind. Koden til det kan ses i Kode Blok 3.

```ts
export const { handlers, signIn, signOut, auth } = NextAuth({
…                                                  { … }
  callbacks: {
…                                                  { … }
    authorized({ auth, request: { nextUrl } }) { ❶
      const protectedPages = ["/settings", "/user", "/forum"];
      const isProtectedPage = protectedPages.some((page) => ❷
        nextUrl.pathname.startsWith(page)
      );
      const isAuthPage = nextUrl.pathname.startsWith("/login") ||
                         nextUrl.pathname.startsWith("/register");
      const isLoggedIn = !!auth?.user;
      const hasRefreshError = auth?.error === refreshTokenError; ❸

      // Handle protected pages
      if (isProtectedPage) { ❹
        if (!isLoggedIn) return Response.redirect(new URL("/login", nextUrl));
        if (hasRefreshError) return Response.redirect(new URL("/login", nextUrl));
        return true;
      }

      // Handle auth pages (login/register)
      if (isAuthPage) {
        if (isLoggedIn && !hasRefreshError) {
          return Response.redirect(new URL("/forum/0", nextUrl)); ❺
        }
        return true;
      }
      return true;
    }
  },
…                                                  { … }
});
```

Kode Blok 3: Autorisation logik (`/auth.ts`, linje 169-204)

Callback funktionen `authorized` ❶ kører ved hver sideanmoding for at afgøre om brugeren skal have adgang til den anmodede side. `auth` parameteret indeholder den nuværende session, og `request` er den anmodning der skal autoriseres. Vi er kun intereseret i `nextUrl` delen, så denne attribut udtrækkes. I funktionen er der defineret en liste af router der kræver autorisation. Der tjekkes om starten af den anmodede URL-sti matcher en af disse, for at

beslutte om det er en beskyttet side ❷. Der tjekkes også om der er sket en fejl i forbindelse med token fornyelse ❸. Hvis der er tale om en beskyttet side, tjekkes der om brugeren er logget ind, og om de har en refresh fejl ❹. Hvis de ikke er logget ind, eller hvis der er fundet en fejl, bliver de omdiregeret til login siden. På samme måde bliver brugerer der er logget ind også omdiregeret til hjem-siden hvis de er på login- eller registerings-siden ❺.

Formular implementation

GNUF anvender adskillige steder formularer til at tage imod bruger input. For effektivt håndtering er der implementeret et robust system med input validering, tilstandsstyring og fejlhåndtering. Til input validering er biblioteket Zod anvendt. Med Zod kan man definere såkaldte "schemas" som bestemmer hvilke regler forskellige felter skal følge for at de kan blive godkendt [24]. Det hjælper blandt andet også med at håndtere fejlmeddelelser for ikke-godkendte felter.

Der vil nu gennemgås et eksempel på et af systemerne til at håndtere formularere. Eksempel fokuserer på systemet til at oprette opslag, men alle formularer følger samme mønster. I Kode Blok 4 kan den første del ses som indeholder et Zod schema, samt opsætningen af interfacet til tilstandsstyring.

```ts
const CreatePostSchema = z.object({ ❶
    title: z ❷
        .string()
        .min(3, {message: 'Post title must be at least 3 characters'})
        .max(300, {message: 'Post title must not be more than 300 characters'}),
    mainText: z
        .string()
        .max(100000, {message: 'Post content must not be more than 100,000
        characters'}),
    communityId: z
        .number()
        .min(1, {message: 'No community chosen'}),
});

export interface CreatePostState { ❸
  ❹ errors?: {
        title?: string[];
        mainText?: string[];
        communityId?: string[];
        image?: string[];
    };
 ❺ fieldsState?: {
        title?: string;
        mainText?: string;
        communityId?: string;
    };
  ❻ message?: string | null;
}
```

Kode Blok 4: Eksempel på formula schema og tilstand (`/lib/actions/createPost.ts`)

Til at starte med opsættes et objekt schema som indeholder de forskellige felter som er inkluderet i formularen ❶. Dette er f. eks. titlen på opslaget (`title`) ❷. For hvert felt bestemmes typen for feltet, samt hvilke regler det skal følge. Fx skal `title` have typen `string`, være mindst tre tegn, og maksimalt 300 tegn. Regler kan også inkludere en tilpasset fejlmeddelelse (`message`).
Dernæst oprettes et tilstands interface til formularen ❸. Formularens tilstand indeholder alle individuelle fejlmeddelelser der skulle have opstået under input validering, kategoriseret efter hvilke felt det er knyttet til ❹. Derudover inkluderer det også tilstanden på hvert felt ❺. Dette bruges til at gendanne felterne, hvis der skulle opstå en fejl, så brugeren ikke skal indtaste alting igen. Til sidst er lagres der også en samlet meddelelse som kan blive vist til brugeren ❻. Den bruges typisk til en overordnet fejlmeddelelse.

Efter opsætningen af dette kommer selve logikken til formularbehandlingen. Et udsnit af koden kan ses i Kode Blok 5.

```ts
export async function createPost(_prevState: CreatePostState, formData: FormData): Promise<CreatePostState> {
    const validatedField = CreatePostSchema.safeParse({ ❶
        title: formData.get('title'),
        mainText: formData.get('mainText'),
        communityId: Number(formData.get('communityId')),
    });

    if (!validatedField.success) { ❷
        return generateFormResponse(formData, validatedField, "Missing or invalid fields");
    }

… { Auth and fetching… }

    if (!response.ok) { ❸
        return generateFormResponse(formData, validatedField, `Error while creating post: ${response.status}`)
    }

    const responseData: CreatePostResponse = await response.json()
… { … }
    redirect(`/post/${responseData.post_id}`) ❹
}
```

Kode Blok 5: Eksempel på formularbehandling (`/lib/actions/createPost.ts`)

Funktionen starter med at validere felterne fra formularen mod vores Zod schema. Dette gøres med `safeParse` metoden ❶. Derefter tjekkes der om det opstod nogle fejl ❷. I tilfælde af en mislykket validering, returneres et objekt i form af `CreatePostState` interfacet fra Kode Blok 4. Efter interaktion med backenden håndteres potentielle fejl, der skulle have opstået ligeledes ❸. Efter en succesfuld oprettelse af et opslag, omdirigeres brugeren til sidst til siden der nu indeholder det ❹.

Backend integrering

Når det kommer til integrering med backend, fungerer dette i GNUF ved at kalde API endpoints via HTTP. Eftersom frontenden modtager data på denne måde, ved TypeScript ikke automatisk hvilke data der er inkluderet eller hvilke typer det er tale om. For at løse dette problem og sikre type-sikkerhed, blev der lavet interfaces for hvert endpoint. Disse interfaces definerer den forventede form af både request og response data. I Kode Blok 6 kan der ses et eksempel på sådan et interface.

```ts
export interface GetPostResponse {
    id: number;
    title: string;
    main_text: string;
    timestamp: string;
    likes: number;
    dislikes: number;
    comment_flag: boolean;
    img?: string;
    author: {
        auth_id: number;
        username: string;
        imagePath: string;
    };
    community: {
        com_id: number;
        name?: string;
    }
}
```

Kode Blok 6: Eksempel på response interface (`/lib/apiTypes.ts` , linje 91-114)

Disse interfaces kan nu bruges til at fortælle TypeScript hvordan formen på et response kommer til at se ud.

Tilpassede React hooks

Et typisk mønster som blev observeret under udviklingen var autentificeret datahenting, hvor endpoint parametre også ændrede sig flere gange i samme komponent. Et eksempel på dette er under henting af opslag, hvor man for at kunne sortere opslag, er nødt til at tilføje et bestemt parameter. Derfor blev der udviklet en genbrugelig React hook til at håndtere dette og abstrahere logikken væk fra alle de komponenter der anvender det mønster. Koden til denne hook kan ses i Kode Blok 7.

```ts
export function useAuthFetch<T>(endpoint: string, dependencies: unknown[] = []) {
  const [result, setResult] = useState<{ ❶
    status: number; data?: T; isLoading: boolean; error?: string;
  }>({ status: 0, isLoading: true });

  const {session} = useCurrentSession(); ❷

  useEffect(() => { ❸
    async function fetchData() {
      if (!session) return;

      try {
        const response = await fetchWithAuth(session, endpoint); ❹
        setResult({ ❺
          status: response.status,
          data: response.data,
          isLoading: false,
        });
      } catch (error) {
        setResult({
          status: 0,
          isLoading: false,
          error: (error as Error).message,
        });
      }
    }

    fetchData();
  }, [endpoint, session, ...dependencies]); ❻

  return result;
}
```

Kode Blok 7: useAuthFetch hook (`/lib/hooks/authFetch.ts`)

Hooket starter med at oprette en state hook `result`, som indeholder alt den data der skal returneres ❶. Den initialiseres med værdien `isLoading` sat til `true`, hvilket kan bruges til at afgøre om resultatet er klart endnu. `useCurrentSession()` anvendes til at få adgang til brugerens JWT fra sessionens data ❷. Derefter bruges `useEffect` til at igansætte en asynkron funktion `fetchData` der fetcher dataet ❸. Det betyder at hooket ikke venter på at `fetchData` er færdig, men derimod fortsætter med bare at returnere indeholdet af `result`.

Inde i `fetchData` bruges `fetchWithAuth` ❹ funktionen, som bare er en general funktion til at kalde endpointet med en JWT og returnere resultatet. Derefter sættes `result` til at indeholde denne data, og `isLoading` bliver sat til `false` ❺. Hooket vil nu returnere det opdaterede `result` i stedet.
Hooket bruger også `useEffekt`, som sørger for at opdatere dataet selv hvis dens container-komponent ikke opdateres. Hver gang `endpoint`, `session` eller nogle yderligere afhængigheder ændres bliver dataet automatisk hentet igen og opdateret ❻.

Implementation af centrale elementer

Eftersom GNUF indeholder et meget stort antal af komponenter er det umuligt at beskrive dem alle i detalje i denne report. Derfor er der valgt at blive fokuseret på et par centrale komponenter som eksempler.

Oprettelse af opslag

Opslag oprettelse foregår igennem et komponent ved navn `CreatePost`. Dette er et ret kompleks komponent som demonstrerer mange af teknikker og elementer som også bruges i andre komponenter. Den yderste del af komponentet kan ses i Kode Blok 8.

```tsx
export default function CreatePost({forumId}: { forumId: string }) {
 return (
   <Dialog> ❶
     <DialogTrigger asChild> ❷
       <Button variant={"outline"}>New Post</Button>
     </DialogTrigger>
     <DialogContent ❸
       includeClose={false}
       onInteractOutside={(event) => event.preventDefault()}
     >
       <DialogHeader>
         <DialogTitle className={"text-2xl"}>New Post</DialogTitle>
       </DialogHeader>
       <div>
         <CreatePostForm forumId={forumId}/> ❹
       </div>
     </DialogContent>
   </Dialog>
 )
}
```

Kode Blok 8: Ydre den af opslag oprettelse komponentet
(`/components/general/createPost.tsx`)

I komponentet gøres der brug af `Dialog` komponentet fra Shadcn ❶. Det er et komponent der skaber en knap som åbner et dialogboks over hjemmesiden når den trykkes på [25]. Den første del er `DialogTrigger` ❷, som er det element som åbner dialogboksen. I eksemplet her er det bare en knap. `DialogContent` ❸ er det som dialogboksen indeholder. Der er inkluderet et par props for at ændre dens adfærd. `includeClose` er en tilpasset prop som blev tilføjet for at

kunne fjerne den lukkeknap som standard er inkluderet i komponentet. `onInteractOutside` bliver ændret til at stoppe dens standard adfærd, som er at lukke dialogen hvis der bliver trykket udenfor boksen. Dette er til for at en bruger ikke ved et uheld lukker boksen og får slettet deres opslag. `CreatePostForm` ❹ er det komponent som indeholder selve formularen til at lave et opslag. Et udsnit af koden til dette komponent kan ses i Kode Blok 9. Bemærk at mange kosmetiske og mindre vigtige elementer er fjernet for at forkusere på de funktionelle aspekter.

```tsx
export default function CreatePostForm({forumId}: { forumId: string }) {
  const [pending, setPending] = useState(false) ❶
  const [formState, dispatch] = useActionState(createPost, {}); ❷
  const [images, setImages] = useState<ImageListType>([]);
                                        { … }
  const handleSubmit = async (formData: FormData) => { ❸
    const imageFile = images[0]?.file;
    if (imageFile) {
      formData.append("image", imageFile);
    }
    dispatch(formData)
  }

  return (
    <form action={handleSubmit} className={"flex flex-col gap-6"}> ❹
      {formState.message && !pending && (
        <div className="text-red-500">{formState.message}</div>
      )}
      <div className={"flex flex-col gap-4 mt-4"}>
                                        { … }
        <CommunitySelect forumId={forumId}/> ❺
                                  { Error messages… }
        <FormInput formState={formState} fieldName={"title"} placeholder={"Title"}
        label={"Title (required)"} inputType={"text" required/> ❼
                          { Image upload and more form fields… }
      </div>
      <div className={"flex justify-between gap-4"}>
        <DiscardButton/> ❻
        <PostButton setPending={setPending}/>
      </div>
    </form>
  );
}
```

Kode Blok 9: Formular-komponent til opslag oprettelse (`components/forms/createPostForm.tsx`, linje 32-141)

Komponentet bruger `useState` til at holde styr på, at formularen er i gang med at blive indsent ❶. `useActionState` er en anden React hook som bruges til at forbinde formularen til den handling den skal fortage når den indsendes ❷. I eksemplet her bruges `createPost` som set tilbage i ***Formular implementation*** (Kode Blok 5). Ud fra denne hook oprettes `formState` som indeholder formularens nuværende tilstand, samt `dispatch` som er funktio-

nen der kaldes når formularen indsendes. Normalt kan `dispatch` kaldes direkte, men i dette komponent skal der inkluderes noget ekstra information udover det der er i formular felterne. Derfor oprettes en funktion `handleSubmit` ❸ som tilføjer ekstra information før `dispatch` kaldes. Denne funktion anvendes som `action` under oprettelsen af formular elementet ❹. Komplekse UI elementer som `CommunitySelect` ❺ og `DiscardButton` ❻ er udtrukket til deres egne komponenter for bedre organisering. Det samme gælder elementer som gentages meget såsom `FormInput` ❻, så de ikke skal oprettes gentagende gange

Bruger profil siden

Det andet eksempel der vil gennemgås er siden der viser brugeres profiler. Det overordnede komponent kan ses i Kode Blok 10.

```tsx
export default function UserPage({params}: Props) {
  const userId = use(params).id; ❶

  const {
    data: userData, isLoading, error
  } = useAuthFetch<GetUserProfileResponse>(`/api/user/profile/${userId}`); ❷

  if (isLoading) return <LoadingSpinner delay={500}/>; ❸

  if (error || !userData) { ❹
    return (
      <div className="p-4 text-center">
        <p className="text-red-500">Failed to load user profile</p>
        <p className="text-sm text-gray-500">{error}</p>
      </div>
    );
  }

  return <UserPageLayout userData={userData}/>; ❺
}
```

Kode Blok 10: Bruger side komponent (`app/(default)/user/[id]/page.tsx`, linje 17-39)

`UserPage` er placeret i en `page.tsx` fil, og er derfor det komponent, der vises til brugeren når de besøger denne side. Siden er placeret i en dynamisk rute der gør at man kan se profiler for forskellige brugere baseret på deres ID. Dette ID udtrækkes med React's `use` hook ❶. Her anvendes `useAuthFetch` hooket (set i Kode Blok 7) til at håndtere API requesten ❷. Komponenten viser en loading indikator når dataen stadig hentes ❸, og viser en fejlmeddelelse hvis der skulle opstå en fejl ❹. Den visuelle del af komponentet er separeret fra datahentningsdelen, og er fundet i et andet komponent `UserPageLayout` ❺. Koden til det komponent kan ses i Kode Blok 11.

```tsx
function UserPageLayout({userData}: { userData: GetUserProfileResponse }) {
  return (
    <div className={`grid grid-cols-4 gap-12 container mx-auto px-20 my-26`}>
      <div className={"flex flex-col gap-6 max-w-70 justify-self-end sticky top-26 h-fit"}>
        <UserInfo userData={userData}/> ❶
        <ForumList/> ❷
      </div>
      <UserPostList userId={userData.id}/> ❸
    </div>
  );
}
```

Kode Blok 11: Layout til bruger side (`app/(default)/user/[id]/page.tsx`, linje 41-51)

`UserPageLayout` indeholder layoutet til siden. De forskellige dele er uddelt til separate komponenter for at holde koden organiseret. Det tillader også progressiv rendering, hvilket betyder at hvert komponent renders så snart det er klart, i stedet for at siden venter til at alle komponenter er klar. Disse komponenter er `UserInfo` ❶ som viser brugerens information, `ForumList` ❷ som skulle vise de fora brugeren er medlem af, og `UserPostList` ❸ som viser de opslag brugeren har oprettet. Kun `UserPostList` vil gennemgås her.

`UserPostList` indeholder et samlet komponent til at vise brugerens opslag, kommentarer, likede opslag og kollektioner. Blandt disse er kun delen der viser brugerens opslag faktisk implementeret. Koden kan ses i Kode Blok 12

```tsx
export default function UserPostList({userId, limit}: Props) {
  const [sortingOption, setSortingOption] = useState<string>("new"); ❶

  return (
    <Card className={`grow relative light-glow-primary col-span-3 min-h-[85vh]`}>
      <Tabs defaultValue="posts" className={`px-10 py-6 gap-4`}> ❷
        <TabsList className={`w-full bg-black/50`}>
          <TabsTrigger value="posts">Posts</TabsTrigger>
          <TabsTrigger value="comments">Comments</TabsTrigger>
          <TabsTrigger value="liked">Liked</TabsTrigger>
          <TabsTrigger value="collections">Collections</TabsTrigger>
        </TabsList>
        <div className="flex items-center justify-between">
          <SortingMenu setCurrentOption={setSortingOption}
          currentOption={sortingOption}/> ❹
        </div>
        <TabsContent value={"posts"}> ❸
          <NormalPostList userId={userId} limit={limit} sortOption={sortingOption}/>
        </TabsContent>
…                          { Other tabs without content… }
      </Tabs>
    </Card>
  );
}
```

Kode Blok 12: UserPostList (`components/userPage/userPostList.tsx`, linje 17-72)

Komponenten indeholder en tilstands variabel til hvordan opslag, og evt. andet indhold, skal sorteres ❶. `Tabs` komponenten fra Shadcn bruges til at organisere indholdet i forskellige faner ❷. Hver fane har en værdi der binder den sammen til et korresponderende `TabsContent` komponent, hvor fanens egentlige indhold er placeret ❸. Der er også et `SortingMenu` ❹ komponent som kontrollerer hvordan indholdet skal sorteres.

I Kode Blok 13 ses koden til det komponent der viser brugerens opslag.

```tsx
function NormalPostList({userId, limit, sortOption}: {userId: number, limit?: number, sortOption: string}) {
  const params = new URLSearchParams(); ❶
  if (limit) {
    params.append("Limit", limit.toString())
  }
  params.append("UserId", userId.toString())

  switch (sortOption) { ❷
    case "top":
      params.append("SortBy", "likes");
      break;
    case "worst":
      params.append("SortBy", "likes");
      params.append("SortOrder", "asc");
  }

  const {
    data: postsData,
    isLoading,
    status,
    error
  } = useAuthFetch<GetMultiplePostsResponse>(`/api/post/posts?${params.toString()}`) ❸
…                              { Loading and error handling… }
  return (
    <div className={"flex flex-col gap-8"}>
      {postsData.posts.map((post) => ( ❹
        <PostThumbnail
          postData={post}
          key={post.post_id}
        />
      ))}
    </div>
  )
}
```

Kode Blok 13: Bruger opslag liste (`components/userPage/userPostList.tsx`, linje 74-118)

Her oprettes et `URLSearchParams` objekt ❶ til at konstruere et URL baseret på forskellige faktorer. Afhængig af hvilke sorterings-filter brugeren anvender, tilføjes der forskellige parametre til URL'et ❷. Dernæst bruges `useAuthFetch` til at hente opslagene baseret på parameterne. Ud fra den hentede data, vises alle opslagene dynamisk ved at mappe API-dataen til komponenter.

8.2 Backend implementering

Intro to DotNet framework

DotNet er en moderne, flerarkitektur-platform udviklet af Microsoft til design af robuste, high-performance backends til web, desktop, mobile, cloud og indlejrede systemer. Den tilbyder en managed runtime, et stort standardbibliotek, og et endnu større økosystem af værktøjer og funktioner til at skrive sikker, vedligeholdbar og effektiv kode. [26]

Til GNUF blev .NET 9.0 brugt, både for dens evne til cross-compilation, så det kunne udvikles på de normale x86_64 computere, men kører på serveren (se Afsnit 8.4). Udover det er der også for ydeevnen, strukturen og renligheden, med dens stærke type safety, indbyggede dependency injection. Endnu en fordel ved DotNet er at den inkluderer Entity Framework Core som standard.

EF Core

Entity Framework Core er en moderne objekt-relationel kortlægger for .NET. Det tillader udviklere at interagere med databaser ved hjælp af C#'s objekter i stedet for rå SQL-forespørgsler, som gør data-adgang mere intuitiv og mindre fejltilbøjelige.

I GNUF agerer EF Core som limen mellem vores C# backend og SQLite databasen. Det kortlægger vores C# klasser (modeller) til database-tabellerne og håndtere at oversætte mellem LINQ-forespørgslerne og SQL under hætten.

Modeller

GnufContext.CS

Formålet med denne klasse er at definere, hvilke modeller der korresponderer med hvilke tabeller i databasen via Entity Framework Core (EF Core). Dette muliggør, at applikationen kan interagere med databasen gennem objektorienteret kode.

```cs
public DbSet<UserStructure> Users { get; set; } = null!;
public DbSet<PostStructure> Post { get; set; } = null!;
public DbSet<CommunityStructure> Community { get; set; } = null!;
public DbSet<FeedbackStructure> Feedback { get; set; } = null!;
```

Kode Blok 14: GnufContext.cs | linje: 10-13

Hver linje erklærer en egenskab i DbContext, der fungerer som en repræsentation af en database-tabel. For eksempel svarer `DbSet<UserStructure> Users` til en tabel (typisk USERS), og gør det muligt at foretage forespørgsler, opdateringer og indsættelser mod denne tabel gennem LINQ eller EF's API.

`DbSet<T>` er en generisk klasse i EF Core, der repræsenterer en samling af entiteter af typen T, som kan behandles som en tabel i databasen.

`{ get; set; }` betyder, at egenskaben kan læses og skrives – hvilket er nødvendigt for at EF kan tracke og ændre data.

`= null!` anvendes for at undertrykke compilerens nullability-advarsler. EF Core initialiserer disse egenskaber ved runtime, ikke i klassens konstruktør, hvilket ellers ville udløse en advarsel i C#'s null-sikkerhedssystem. Udråbstegnet (null-forgiving operator) fortæller compileren, at null ikke vil forekomme ved brug.

Dette setup er en kernekomponent i EF Core's ORM-arkitektur og gør det muligt at koble klasser i C# direkte til relationelle databasetabeller.

Entity Models (EMs)

Formålet med Entity Models er at repræsentere relationelle databasetabeller som objektorienterede klasser via EF Core. Disse klasser anvendes som datalagets strukturelle rygrad, og er typisk lokaliseret i roden af `model/` mappen. De kortlægges direkte til databasen via attributer som `[Table]`, `[Column]`, og `[Key]`. Dermed kan EF Core automatisk håndtere CRUD-operationer (Create, Read, Update, Delete) og relationer mellem entiteter. Et eksempel på sådan en EM kan ses nedenfor:

```cs
// Models/User.cs
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Gnuf.Models;

[Table("USER")]
public class UserStructure
{
    [Key]
    [Column("USER_ID")]
    public int UserId { get; set; }

    ...

    [Required]
    [Column("USERNAME")]
    [MaxLength(100)]
    public string Username { get; set; } = string.Empty;

    [Required]
    [Column("PASSWORD")]
    [MaxLength(1000)]
    public string Password { get; set; } = string.Empty;

    [Required]
    [Column("SALT")]
    public string Salt { get; set; } = string.Empty;

    ...

    [Required]
    [Column("ADMIN")]
    public int IsAdmin { get; set; } = 0;

    ...
}
```

Kode Blok 15: Models/User.cs

Bemærk: Feltet IsAdmin er eksplicit initialiseret til 0, da den typiske bruger ikke er en administrator. Dette bruges som et "flag" og kan ændres til 1 for at give administratorrettigheder.

DTO (Data Transfer Objects)

Disse findes i undermapperne f.eks `/Models/User` og bruges til at strukturere data, der modtages eller returneres via API'et. DTO'er gør det muligt at adskille den underliggende datamodel fra den datarepræsentation, der eksponeres i HTTP-requests og -responses. I dette tilfælde gør det det muligt at have HTTP i JSON format, men have den interne struktur i klasser, som er markant nemmere at arbejde med i henhold til EF Core. Et eksempel på sådanne DTO kan ses nedenfor:

```cs
// Models/User/GetUserProfile.cs
namespace Gnuf.Models.User
{
    public class GetUserProfileResponse
    {
        public int? Id { get; set; }
        public string Email { get; set; }
        public string Username { get; set; }
        public string? ImgPath { get; set; }
        public int? PostIds { get; set; }
        public int? CommunityIds { get; set; }
        public int? Tags { get; set; }
        public int? Admin { get; set; }
    }
}
```

Kode Blok 16: Models/User/GetUserProfile.cs

Bemærk: Navneområdet (namespace) er her udvidet til Gnuf.Models.User i stedet for blot Gnuf.Models. Dette skyldes, at DTO'en er et "submodul" knyttet til en specifik funktionalitet (i dette tilfælde brugerprofilen), og derfor organiseres i en undermappe. Dette giver en mere overskuelig og modulær projektstruktur.

Services

Services er specialiserede klasser, der indkapsler logik og funktionalitet, som hverken hører til modeller eller controllere. Services leveres til controllere som har brug for dem, og derved kan alle controllere få adgang til den funktionalitet som servicen tilbyder. På den måde adskilles anmodningshåndtering fra den bagvedliggende logik, og derved gøres koden mere organiseret. I GNUF eksisterer der en enkelt service, som bliver brugt til håndtering af JWT'er. Logikken til dette er relativt kompleks, så derfor gav det mening at separere det fra resten.

Tokenhåndtering

Som tidligere nævnt under ***Autentifikation og autorisation*** i Afsnit 8.1, anvendes der i GNUF et JWT-baseret autentifikations system. Generering og validering af disse tokens sker med hjælp af servicen `TokenService`. Selve generering af JWT'er foregår igennem `GenerateJwtAccessToken` metoden som set i Kode Blok 17.

```cs
public class TokenService : ITokenService
{
                                    { config setup… }
    public string GenerateJwtAccessToken(UserStructure user)
    {
        var jwtKey = _configuration["Jwt:Key"];
        var jwtIssuer = _configuration["Jwt:Issuer"];
        var jwtAudience = _configuration["Jwt:Audience"];
        var jwtExpireMinutes = int.Parse(_configuration["Jwt:ExpireMinutes"] ?? "60");

        var claims = new List<Claim> ❶
        {
            new(JwtRegisteredClaimNames.Sub, user.UserId.ToString()),
            new(JwtRegisteredClaimNames.Name, user.Username),
            new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
            new(ClaimTypes.Role, user.IsAdmin == 1 ? "Admin" : "User"),
            new(JwtRegisteredClaimNames.Email, user.Email),
            new("imagePath", user.ImagePath ?? "")
        };

        var key = new SymmetricSecurityKey( ❷
            Encoding.UTF8.GetBytes(jwtKey ?? throw new InvalidOperationException("JWT
            Key is not configured")));
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256); ❸

        var token = new JwtSecurityToken( ❹
            jwtIssuer,
            jwtAudience,
            claims,
            expires: DateTime.Now.AddMinutes(jwtExpireMinutes),
            signingCredentials: creds
        );
        return new JwtSecurityTokenHandler().WriteToken(token);
    }

                        { functions for validation and refreshtokens… }
}
```

Kode Blok 17: Udbyder-konfiguration og login logik (`/auth.ts` linje 69-112)

Til konstruktionen af JWT'en, oprettes en liste af forskellige claims som hver token skal indeholde ❶. Dette er eksempelvis bruger-id og autorisationsniveau, og bruges til at verificere at brugeren er hvem de udgiver sig for at være. Systemet bruger en symmetrisk nøgle[3] til at

[3]En hemmelig værdi som kun serveren kender

signere tokens ❷. Denne nøgle pakkes ind i et `SigningCredentials` objekt sammen med den algoritme der skal bruges til at signere JWT'en, i dette tilfælde HMAC-SHA256 ❸. Når den genereres ❹ vil dens signatur være baseret på dens header, payload og den symmetriske nøgle, alt sammen hashet ved brug af den valgte algoritme. Under validering af en token, genberegnes denne signatur så for at se om den passer. Derved kan det verificeres at JWT'en er autentisk, og ikke er blevet ændret på nogen måde.

Servicen indeholder yderligere to metoder som ikke vil gennemgås i dybden. Den ene metode validere tokens, som beskrevet førhen. Den anden anden metode står for at generere refresh tokens, som er længerevarende tokens med minimale krav. Sammen med en adgangstoken bruges de til at generere nye adgangstokens uden at brugeren skal logge ind igen.

Controllere

Controllere er bindeleddet mellem frontend og backend. Det er disse som definerer alle API-endpoints, tager imod HTTP-Requests, kalder den nødvendige logik (f.eks database opslag) og sender svar tilbage i form a JSON eller HTTP koder.

For et detaljeret indblik i database tabellerne, samt JSON formattet som bliver brugt til parsing mellem FE og BE, se [27]

AuthController (4.1)

Controlleren håndterer al logik i forhold til autorisation, heriblandt login & register logik, password hashing, samt tokens til user sessions.

Class setup

```cs
[ApiController]
[Route("api/[controller]")]
…
public class AuthController : ControllerBase
{
    private readonly IConfiguration _configuration;
    private readonly GnufContext _context;
    private readonly ITokenService _tokenService;

    public AuthController(GnufContext context, ITokenService tokenService,
    IConfiguration configuration)
    {
        _context = context;
        _tokenService = tokenService;
        _configuration = configuration;
    }
```

Kode Blok 18: /Controllers/AuthController.cs | Linje 15-28

`[ApiController]` fortæller ASP.NET Core at denne klasse skal behandles som en API-Controller. Dette aktiverer en lang række smarte funktioner, som for eksempel modelvalidering (returnering af HTTP kode 400 ved forkert request), binding af JSON og DTO's automatisk 415/400 HTTP-håndtering, m.m. [27]

[Route("api/[controller]")] definerer root URL for denne controller. I dette tilfælde er det `api/auth`. Alle endpoints vil så blive sat efter dette, f.eks `api/auth/login`.

`public class AuthController : ControllerBase`

- Klassen AuthController er controlleren for autentificering (login, registrering osv.).
- Den nedarver fra ControllerBase, som er en lettere version af Controller uden view-support. Perfekt til REST APIs hvor man kun returnerer JSON og ikke arbejder med HTML-sider.

```cs
private readonly IConfiguration _configuration;
private readonly GnufContext _context;
private readonly ITokenService _tokenService;
```

Disse tre objekter bliver injiceret via dependency injection i constructoren:

- GnufContext: Databasekonteksten, bruges til at læse og skrive til databasen.
- ITokenService: Servicen brugt til at generere og validere JWT-tokens (login-tokens), som tidligere set i afsnittet ***Tokenhåndtering***.

- IConfiguration: Indeholder indstillinger fra appsettings.json og miljøvariabler, f.eks. hemmelige nøgler eller token-levetid.

Helper functions

Adgangskoder opbevares ikke i klartekst i databasen. I stedet bliver de hashed med en kryptografisk algoritme. Dette er en grundlæggende sikkerhedspraksis, som beskytter brugernes data i tilfælde af et databreach.

I GNUF anvendes Argon2id, som på nuværende tidspunkt (2025) betragtes som den mest sikre og moderne algoritme til password hashing, ifølge adskillige online kilder, som f.eks OWASP [28]. Argon2id er designet til at være både tidskrævende og RAM-krævende, hvilket gør det ekstremt svært at udføre brute-force angreb, særligt i stor skala.

```cs
    // Helper: Generate random salt
    private byte[] GenerateSalt(int length = 16)
    {
        using var rng = RandomNumberGenerator.Create();
        var salt = new byte[length];
        rng.GetBytes(salt);
        return salt;
    }
```

Kode Blok 19: /Controllers/AuthController.cs | Linje 31-37

Denne helper-metode genererer et kryptografisk sikkert salt – en tilfældig byte-array (standardlængde: 16 byte). Saltet sikrer, at to identiske passwords ikke vil resultere i samme hash. Dermed forhindres brugen af prækompilerede hash-tabeller, hvilket er en metode hvorved angribere vil generere hashes over de f.eks 1000 mest populære passwords, og se om nogen af dem matcher.

```cs
// Helper: Hash password using Argon2id
private async Task<string> HashPasswordAsync(string password, byte[] salt)
{
    var argon2 = new Argon2id(Encoding.UTF8.GetBytes(password))
    {
        Salt = salt,
        DegreeOfParallelism = 4, // threads
        MemorySize = 65536, // 64 MB
        Iterations = 4
    };

    var hash = await argon2.GetBytesAsync(32); // 256-bit hash
    return Convert.ToBase64String(hash);
}
```

Kode blok 20: /Controllers/AuthController.cs | Linje 40-52

Denne funktion konfigurerer og kører selve hash-processen:

- Salt: Det tilfældigt genererede salt bruges som input.

- DegreeOfParallelism: Tillader brug af 4 tråde for at fremskynde beregningen.
- MemorySize: Sætter RAM-forbruget til 64 MB per hash. Dette er uproblematisk ved bruger-login, men en massiv udfordring for angribere, der forsøger at hash mange passwords samtidigt.
- Iterations: Antallet af gange hash-processen gentages (her: 4) – øger tiden og kompleksiteten yderligere.

`var hash = await argon2.GetBytesAsync(32);` Returnerer et 256-bit (32 bytes) hash.

Til sidst returneres `Convert.ToBase64String(hash);` hvilket konverterer byte-array'en til en Base64-string, hvilket gør det nemt at gemme i databasen.

```cs
// Helper: Sets a refresh token in cookies
private void SetRefreshToken(string refreshToken)
{
    var options = new CookieOptions
    {
        HttpOnly = true,
        Expires = DateTime.UtcNow.AddDays(int.Parse(_configuration["RefreshToken:ExpireDays"] ?? "7")),
        Secure = true,
        SameSite = SameSiteMode.Strict,
        Path = "/api/auth"
    };

    Response.Cookies.Append("refreshToken", refreshToken, options);
}
```

Kode blok 21: /Controllers/AuthController.cs | Linje 61-74

Generere en refreshtoken, som kan bruges af brugeren til at få en ny session token inden for 7 dage, så de ikke behøver logge ind igen hver gang.

Refresh token bliver opbevaret lokalt i browseren som en cookie med parametrene:
- HttpOnly; Betyder at cookiens indhold ikke kan tilgås fra JavaScript, og er dermed immun over for f.eks XSS-angreb
- Expires; Sætter cookien til at udløbe efter 7 dage, for at begrænse skaden, både hvis enhed / cookie er stjålet, men ligeledes hvis brugeren er logget ind på f.eks et offentligt bibliotek.
- Secure: Betyder at cookien kun vil blive sendt over HTTPS, altså krypterede forbindelser. Dette betyder at en "sniffer" ikke vil kunne stjæle cookien på et offentligt netværk, og tilgå din profil således. En sniffer er en som læser eller "sniffer" netværks traffiken for adgangskoder, cookies, og andet som kan misbruges i et cyber angreb. Dette sker oftest på offentlige, usikrede netværk
- SameSite; afgører hvornår cookien kan blive sendt af browseren. Hvis dette var en tracker, for adbrokers eller lign. ville vi have valgt at sende den på flere sider såsom Facebook, men da dette er adgangs token til kontoen, er cookien i "strict mode", hvilket betyder at den kun må blive sendt på samme domæne som den kom fra (i dette tilfælde gnuf.online)

- Path; Hvilken URL-sti cookien skal sendes til. Dette er primært et organisatorisk parameter, og viser hvor cookien helt præcist skal sendes til. Den tilbyder dog også minimal sikkerhed gennem scope-begrænsning, da det forhindrer at cookien sendes til offentlige dele af hjemmesiden, hvor de potentielt kunne blive eksponeret.

User Login

```cs
var user = await _context.Users
  .FirstOrDefaultAsync(u => u.Username == request.Username);

if (user == null) return Unauthorized("Invalid credentials");
```

Kode blok 22: /Controllers/AuthController | Linje 102-105

Her forsøges brugeren slået op i Users-tabellen baseret på det angivne brugernavn. Returneres null, betyder det, at brugeren ikke findes – og der returneres en HTTP 401 Unauthorized. Fejlmeddelelsen er bevidst vag ("Invalid credentials") af sikkerhedsmæssige årsager: Man må ikke kunne konkludere, om det er brugernavnet eller adgangskoden der er forkert – ellers vil angribere kun skulle gætte ét af to.

```cs
var salt = Convert.FromBase64String(user.Salt ?? "");

if (!await VerifyPasswordAsync(request.Password, salt, user.Password))
  return Unauthorized("Invalid credentials");
```

Kode blok 23: /Controllers/AuthController | Linje 107-110

`user` er et objekt som indeholder alle brugerens attributter, og det er derfor nemt at tjekke password. Når brugeren er fundet, hentes det tilhørende salt, som tidligere blev gemt under registreringen. Derefter genberegnes hash'en af det angivne password ved hjælp af samme algoritme og salt (Argon2id). Hvis det resulterende hash ikke matcher det gemte i databasen, afvises loginforsøget med samme generiske fejlmeddelelse som før.

```cs
var accessToken = _tokenService.GenerateJwtAccessToken(user);
var refreshToken = _tokenService.GenerateJwtRefreshToken(user);

SetRefreshToken(refreshToken);

return Ok(new { user.UserId, user.Username, user.Email, user.ImagePath, user.IsAdmin,
accessToken });
```

Kode blok 24: /Controllers/AuthController | Linje 112-117

Ved succesfuld login genereres to tokens:

- Access token: Et kortvarigt JSON Web Token (JWT), som sendes med i efterfølgende requests for at bevise brugerens identitet. Det har en levetid på typisk 1 time og gemmes client-side.
- Refresh token: Et langtids-token der bruges til at få et nyt access token, uden at brugeren behøver logge ind igen. Dette gemmes som en HttpOnly-cookie og udløber som standard efter 7 dage.

Til sidst returneres brugerens oplysninger sammen med access token'et, så frontend kan bruge det til at initialisere sessionen.

Create Post

`if (!User.MatchesId(post.auth_id.ToString())) return Unauthorized();`

Sikrer at bruger id i token er den samme som den i request, for at forhindre at man kan poste som andre brugere

caption: "/Controllers/PostController.cs | Linje xx-xx", kind: "code", supplement: "Kode blok")

Dette endpoint bruges til at oprette et nyt indlæg i GNUF-platformen. Det kan både være en original post eller en kommentar til et andet indlæg (angivet med `comment_flag` og `post_id_ref`).

```cs
if (!User.MatchesId(post.auth_id.ToString())) return Unauthorized();
```

Kode blok 25: /Controllers/PostController.cs | Linje 40

Her valideres det, at `auth_id` i den indkomne request matcher den bruger, som er logget ind (ud fra token claims). Dette beskytter mod spoofing, hvor en bruger kunne forsøge at poste som en anden.

```cs
var newPost = new PostStructure
{
    Title = post.Title,
    MainText = post.MainText,
    auth_id = post.auth_id,
    com_id = post.com_id,
    post_id_ref = post.post_id_ref,
    comment_flag = post.comment_flag,
    timestamp = DateTime.UtcNow,
    likes = 0,
    dislikes = 0,
    comment_Count = 0,
    comments = "",
    Img = post.Img
};
```

Kode blok 26: /Controllers/PostController.cs | Linje 42-56

Et nyt `PostStructure`-objekt bliver instansieret med de nødvendige felter:

- `timestamp` sættes til nuværende UTC-tidspunkt for konsistens.
- `likes`, `dislikes` og `comment_Count` sættes til 0, da posten er nyoprettet.
- `comments` sættes til en tom CSV-streng som forberedelse til mulige fremtidige kommentarer.

```cs
var author = await _context.Users.FirstOrDefaultAsync(u => u.UserId == post.auth_id);
if (author == null) return NotFound();
```

Kode blok 27: /Controllers/PostController.cs | Linje 59-63

Herefter tjekkes det, om brugeren der forsøger at poste, findes i databasen. Hvis ikke, returneres HTTP 404 Not Found.

```cs
_context.Post.Add(newPost);
await _context.SaveChangesAsync();

author.PostIds += string.IsNullOrWhiteSpace(author.PostIds)
    ? $"{newPost.PostID}"
    : $",{newPost.PostID}";
```

Kode blok 28: /Controllers/PostController.cs | Linje 66-72

Den nye post gemmes i databasen, og brugerens `PostIds`-felt opdateres. Post-ID'et bliver tilføjet til en kommasepareret liste, som bruges til at holde styr på hvilke indlæg en bruger har lavet.

```cs
if (newPost.comment_flag)
{
    var parentPost = await _context.Post.FindAsync(newPost.post_id_ref);
    if (parentPost == null)
    {
        return NotFound("Parent post not found");
    }

    parentPost.comments += string.IsNullOrWhiteSpace(parentPost.comments)
        ? $"{newPost.PostID}"
        : $",{newPost.PostID}";

    parentPost.comment_Count++;
}
```

Kode blok 29: /Controllers/PostController.cs | Linje 75-88

Hvis det nye indlæg er en kommentar (angivet med `comment_flag == true`), sker følgende:

- Den oprindelige post (`post_id_ref`) slås op i databasen.
- Findes den ikke, returneres HTTP 404 med beskeden `"Parent post not found"`.
- Kommentar-ID'et tilføjes til forælderens `comments` CSV-streng.
- Antallet af kommentarer (`comment_Count`) inkrementeres.

```cs
return Ok(new { post_id = newPost.PostID });
```

Kode blok 30: /Controllers/PostController.cs | Linje 92

Ved succes returneres HTTP 200 OK samt det oprettede `post_id`. Dette ID kan bruges af frontend til at vise posten eller tilføje den til feedet.

Sikkerhedsanmærkning: Endpointet beskytter mod spoofing ved at validere token-oplysninger mod `auth_id`. Det sikrer ligeledes dataintegritet ved at kræve eksisterende bruger og korrekt referencepost for kommentarer.

Get Post

```cs
[HttpGet("view/{post_id}")]
public async Task<ActionResult> GetPost(int post_id)
```

Kode blok 1: /Controllers/PostController.cs | Linje 96-97

Dette endpoint returnerer alle relevante oplysninger om et specifikt indlæg, inklusive tilhørende bruger- og communityinformation. Hvis brugeren er logget ind, returneres desuden brugerens stemmetilstand (like/dislike/neutral) på det pågældende indlæg.

```cs
var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
var user = await _context.Users.FindAsync(Convert.ToInt32(userId));
```

Kode blok 31: /Controllers/PostController.cs | Linje 99-100

Brugerens ID udtrækkes fra JWT-tokenets claims, og brugeren hentes fra databasen. Dette bruges senere til at bestemme brugerens stemmetilstand på det pågældende indlæg (via `LikeId` og `DislikeId`).

```cs
var matchingPost = await (from post in _context.Post
                          join author in _context.Users on post.auth_id equals author.UserId
                          join community in _context.Community on post.com_id equals community.CommunityID
                          where post.PostID == post_id
                          select new
                          {
                              id = post.PostID,
                              title = post.Title,
                              main_text = post.MainText,
                              post.timestamp,
                              post.likes,
                              post.dislikes,
                              post.post_id_ref,
                              post.comment_flag,
                              post.Img,
                              comment_count = post.comment_Count,
                              post.comments,
                              VoteState = GetVoteState(post_id.ToString(), user.LikeId, user.DislikeId),
                              author = new
                              {
                                  post.auth_id,
                                  author.Username,
                                  author.ImagePath,
                                  author.IsAdmin,
                              },
                              community = new
                              {
                                  post.com_id,
                                  community.Name,
                              }
                          }).FirstOrDefaultAsync();
```

Kode blok 32: /Controllers/PostController.cs | Linje 103-134

En kombineret forespørgsel (via LINQ) bruges til at returnere alle nødvendige data i én samlet anonym struktur. Forespørgslen slår op i tre tabeller:

- `Post` : selve indlægget
- `Users` : brugeren, som har oprettet indlægget
- `Community` : det community, som indlægget hører til

Felter i det returnerede objekt inkluderer:

- `id` , `title` , `main_text` , `timestamp` , `likes` , `dislikes` : kernedata for indlægget.
- `post_id_ref` , `comment_flag` : metadata for kommentarer og relationer til andre indlæg.
- `comments` , `comment_count` : en CSV-liste over kommentar-ID'er og det totale antal kommentarer.
- `Img` : sti til eventuelt billede vedhæftet indlægget.
- `VoteState` : bestemmer om brugeren har liket, disliket eller ikke stemt.

- `author`: indeholder oplysninger om forfatteren (bl.a. `Username`, `ImagePath`, og `IsAdmin`).
- `community`: returnerer det community, som posten tilhører.

Stemmetilstanden (`VoteState`) afgøres ved at sammenligne postens ID med brugerens `LikeId` og `DislikeId`. Dette gøres for at vise korrekt visuel feedback på frontend (f.eks. hvilken knap der skal være aktiv).

```cs
if (matchingPost == null)
{
    return NotFound();
}
return Ok(matchingPost);
```

Kode blok 33: /Controllers/PostController.cs | Linje 136-142

- Hvis intet match findes, returneres HTTP 404.
- Ellers returneres hele det sammensatte JSON-objekt med HTTP 200 OK.

Sikkerhedsanmærkning: Selvom der her returneres data om forfatteren, eksponeres der ikke følsomme oplysninger som email eller adgangsniveau. Desuden er stemmeinformationer afhængige af aktiv token, så kun autentificerede brugere ser deres stemmetilstand.

Get Multiple Posts

```cs
var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
var user = await _context.Users.FindAsync(Convert.ToInt32(userId));
```

Kode blok 34: /Controllers/PostController.cs | Linje 200-201

Bruges til at bestemme VoteState for hvert opslag – altså om brugeren har liket, disliket eller ikke stemt.

```cs
try
        {
            if (query.TimestampStart.HasValue)
                timestampStart =
                DateTimeOffset.FromUnixTimeSeconds(query.TimestampStart.Value).UtcDateTi

            if (query.TimestampEnd.HasValue)
                timestampEnd =
                DateTimeOffset.FromUnixTimeSeconds(query.TimestampEnd.Value).UtcDateTime
        }
        catch (ArgumentOutOfRangeException ex)
        {
            // return a 400 Bad Request instead of crashing the whole request
            return BadRequest("Invalid Unix timestamp provided.");
        }
```

Kode blok 35: /Controllers/PostController.cs | Linje 206-218

Hvis TimestampStart eller TimestampEnd er angivet i Unix-format, konverteres de til DateTime. En try/catch-blok fanger ugyldige timestamps og returnerer en HTTP 400 Bad Request i stedet for at crashe hele kaldet.

```
var postsQuery = _context.Post.AsQueryable();
```

Kode blok 36: /Controllers/PostController.cs | Linje 220

Kode blok 36 initialiser Post objektet som queryable

```cs
postsQuery = postsQuery.Where(p => p.comment_flag == query.GetComments || !
p.comment_flag == query.GetPosts);

        if (query.ParentPostId.HasValue)
        {
            postsQuery = postsQuery.Where(p => p.post_id_ref ==
            query.ParentPostId.Value);
        }

        if (query.CommunityId.HasValue)
            postsQuery = postsQuery.Where(p => p.com_id == query.CommunityId.Value);

        if (query.UserId.HasValue)
            postsQuery = postsQuery.Where(p => p.auth_id == query.UserId.Value);

        if (query.TimestampStart.HasValue)
            postsQuery = postsQuery.Where(p =>
                p.timestamp >=
                DateTimeOffset.FromUnixTimeSeconds(query.TimestampStart.Value).UtcDateTi

        if (query.TimestampEnd.HasValue)
            postsQuery = postsQuery.Where(p =>
                p.timestamp <=
                DateTimeOffset.FromUnixTimeSeconds(query.TimestampEnd.Value).UtcDateTime
```

Kode blok 37: /Controllers/PostController.cs | Linje 222-241

Kode blok 37 tilføjer filtrer til post query, såsom kommentar, ParentPostID, CommunityID, UserID og TimeStamps

```cs
// Sorting
postsQuery = query.SortBy.ToLower() switch
{
    "likes" => query.SortOrder == "asc"
        ? postsQuery.OrderBy(p => p.likes - p.dislikes)
        : postsQuery.OrderByDescending(p => p.likes - p.dislikes),
    "comments" => query.SortOrder == "asc"
        ...
    _ => query.SortOrder == "asc"
        ...
};
```

Kode blok 38: /Controllers/PostController.cs | Linje 243-255

Resultaterne fra querien sorteres så efter enten likes, commentarer eller nyeste, som set på Kode blok 38. OBS: kun likes og tid er faktisk brugt, commentar sortering bliver aldrig brugt

```cs
var limit = Math.Clamp(query.Limit, 1, 100);
var offset = Math.Max(query.Offset, 0);
```

Kode blok 39: /Controllers/PostController.cs | Linje 258-259

Kode blok 39 bruges til pagination. Limit definerer hvor mange opslag vi maksimalt vil have returneret, og offset definerer hvor mange posts den skal springe over før den skal begynde returnere. Dette offset ville blive brugt til en content recommendation algorithm, men der var desværre ikke tid til at lave dette, og derfor er dette offset, unused

```cs
var matchingPosts = await (from post in postsQuery
    join author in _context.Users on post.auth_id equals author.UserId
    join community in _context.Community on post.com_id equals community.CommunityID
    select new
    {
        ...
    })
.Skip(offset)
.Take(limit)
.ToListAsync();
```

Kode blok 40: /Controllers/PostController.cs | Linje 261-293

Ved hjælp af LINQ joins inkluderes info om forfatter og community i hvert opslag, som set på Kode blok 40. Der returneres bl.a.:

- Titel, tekst, tid, likes, dislikes
- Stemmetilstand (VoteState) ift. aktuel bruger

Forfatter: navn, billede, adminstatus

Community: navn og ID

```cs
return Ok(new {
    posts = matchingPosts,
    total_count = totalCount,
    next_offset = offset + matchingPosts.Count
});
```

Kode blok 41: /Controllers/PostController.cs | Linje 295-300

posts: Liste over opslag

total_count: Antal opslag der matcher forespørgslen

next_offset: Bruges af frontend til næste paginerede kald (unused)

Get multiple post by IDs

funkere rimelig ens til ***Get Multiple Posts*** med forskellen værende at den modtager en csv formateret string af IDs som den så loader posts ud fra

```cs
var postsRaw = await _context.Post
          .Where(p => postIdList.Contains(p.PostID)  && p.comment_flag != true)
          .Take(data.limit)
          .ToListAsync(); // <-- Fetch raw entities first
```

Kode blok 42: /controllers/PostController.cs | Linje 321-324

8.3 Database implementering

En database er standard metoden til at gemme, organisere og håndtere data på. Den kan bruges i brede samt både store og små systemer, da den muliggør effektiv lagring, søgning og manipulation af store datamængder på en struktureret måde.

Database Schema

Database Schema er en måde at forklare hvordan databasens struktur fungerer, f. eks. under ***User*** findes *Column name*, *Type*, *Constraints* og *Description*. *Column name* referer til navnet på den kolonne i databasen, også refereret til i Kode Blok 15. *Type* refererer til kolonnens datatype, f. eks. ID i ***User*** med typen INT. *Contraint* refererer til rammerne, som kolonnen skal begrænses til. Ved CHECK ([Condition]) tjekkes om værdien i kolonnen opfylder et krav (f. eks. at USERNAME skal indeholde mindre end 100 tegn). *Description* forklarer hvad de forskellige kolonner indebærer. Tabel 6 viser strukturen på db tabellen på Kode Blok 15.

User

Column Name	Type	Constraints	Description
ID	INT	PRIMARY KEY	Unique Identifier
EMAIL	TEXT	UNIQUE. NOT NULL	User's Email
USERNAME	TEXT	UNIQUE. NOT NULL. CHECK (LENGTH(USERNAME) < 100)	Unique Username
PASSWORD	TEXT	NOT NULL. CHECK (LENGTH(PASSWORD) < 1000)	Hashed password (argon2)
SALT	TEXT	NOT NULL	Hashing salt (argon2)
IMG_PATH	TEXT		URL TO PP
POST_IDs	str[csv]		Array of post IDs (FK to posts.id)
LIKE_IDs	str[csv]		Array of post IDs (FK to posts.id)
DISLIKE_IDs	str[csv]		Array of post IDs (FK to posts.id)
COMMENT_IDs	str[csv]		Array of post IDs (FK to posts.id)
COMMUNITY_IDs	str[csv]		Array of community IDs and names [id. name. id. name.] (FK to communities.id) (FK to communities.name)
ADMIN	BOOL	NOT NULL. DEFAULT FALSE	ADMIN FLAG
TAGS	str[csv]		Array of tags for content recommendation

Tabel 6: user structure

dette tabel viser det db table som i ser strukturen til i Kode Blok 15.

8.4 Serverarkitektur

Serveren der driver GNUF er en Raspberry Pi 5 (8GB model), overclocket til 3,1GHz for at give ekstra ydeevnemargin. Dette er afgørende for pålideligt at hoste en offentligt tilgængelig social medie platform.

Både frontend- og backend-komponenterne krydskompileres til ARM64 og overføres til serveren. De administreres af to Systemd tjenester, som sikrer at begge applikationer automatisk starter ved opstart og genstarter sig selv i tilfælde af nedbrud eller kritiske fejl.

Systemd

Systemd er standardprogrammet til styring af tjenester på de fleste moderne Linux-distributioner, herunder Rasbperry Pi OS (som er baseret på Debian). Den håndterer baggrundsprocesser, også kaldet "daemons", som er essentielle for systemets drift. Disse kan omfatte netværk, enhedsdrivere, logning og mere. [29]

Ved at konfigurere systemd-tjenester for både frontend og backend opnår vi flere fordele:

- Automatisk opstart ved systemstart
- Vedvarende logning
- Watchdog-funktionalitet for at opdage manglende respons og genstarte processer efter behov.

Denne tilgang sikrer at serveren forbliver stabil og responsiv, selv i tilfælde af uventede fejl.

Offentlig Hosting

For at GNUF kunne anvendes i T2 (se Afsnit 9.3), skulle det være offentligt tilgængeligt. For at muliggøre dette blev domænet "gnuf.online" registreret.

Til at håndterdete DNS og yde beskyttelse (fx mod DDoS-angreb) anvendes Cloudflare. Men eftersom campusnetværket begrænser direkte port forwarding, blev siden implementeret ved hjælp af Cloudflare Zero Trust. Dette skaber en sikker, udgående tunnel fra vores server til Cloudflares edge-netværk, og derved elimineres behovet for traditionel port forwarding, mens siden er offentligt tilgængelig via Cloudflares infrastruktur.

Netværksadministration

For at forenkle administrativ adgang, blev et isoleret lokalt netværk oprettet. Gateway-enheden tilsluttes universitetets Ethernet og etablerer både et LAN og et WLAN, hvilket danner et dedikeret netværksmiljø. Dette giver autoriserede administratorer mulighed for direkte at oprette forbindelse og SSH'e ind på Raspberry Pi'en uden at skulle eksponere interne tjenester for det bredere campusnetværk.

9 Testing

Under udviklingen af Gnuf blev der fortaget flere forskellige tests for at sikre applikationens funktionalitet. I dette afsnit vil pågældende tests og deres resultater beskrives.

9.1 Test 0

Den første udførte test var en integrationtest, hvor alle de forskellige backend endpoints blev tested for at se om de fungerede som de skulle. Denne test blev lavet følgende gange som endpoints blev udviklet og opdateret, for at sikre at de fortsat virkede. Resultaterne fra denne test kan ses i Tabel 7.

Kategori	Specksheet Ref	Metode	Endepunkt	Beskrivelse	Implementeret	Test resultat
Autentificering	4.1.1	POST	/api/auth/login	Autentificer bruger og returnér adgangstoken	Implementeret, i brug	Virker
Autentificering	4.1.2	POST	/api/auth/register	Registrér ny bruger og returnér adgangstoken	Implementeret, i brug	Virker
Brugerhåndtering	4.2.1	GET	/api/user/profile/{user_id}	Hent brugerprofil	Implementeret, i brug	Virker
Brugerhåndtering	4.2.2	DELETE	/api/user/remove/{user_id}	Slet brugerprofil	Implementeret, ikke i brug	Virker
Brugerhåndtering	4.2.3	PUT	/api/user/update/user/{user_id}	Opdater brugerens profil (selv)	Ikke implementeret	Ikke testet
Brugerhåndtering	4.2.4	PUT	/api/user/update/backend/{user_id}	Opdater backend-data for bruger	Implementeret, i brug	Virker
Fællesskab	4.3.1	POST	/api/community/create	Opret et nyt fællesskab	Implementeret, i brug	Virker
Fællesskab	4.3.2	GET	/api/community/details/{community_id}	Hent detaljer for et fællesskab	Implementeret, i brug	Virker
Fællesskab	4.3.3	PUT	/api/community/update/details/{community_id}	Opdater fællesskabsdetaljer (admin)	Implementeret, i brug	Virker
Fællesskab	4.3.4	PUT	/api/community/update/backend	Opdater medlemstal, tags og post-ID'er	Implementeret, i brug	Virker
Fællesskab	4.3.5	DELETE	/api/community/remove/{community_id}	Slet fællesskab	Implementeret, i brug	Virker
Fællesskab	4.3.6	GET	/api/community/all	Hent alle fællesskaber (ID, navn, beskrivelse)	Implementeret, i brug	Virker
Post	4.4.1	POST	/api/post/create	Opret et nyt Post	Implementeret, i brug	Virker
Post	4.4.2	GET	/api/post/view/{post_id}	Hent det,Virker"aljer for Post"	Implementeret, i brug	Virker
Post	4.4.3	PUT	/api/post/update/user/{post_id}	Rediger Post (bruger)	Implementeret, ikke i brug	Virker
Post	4.4.4	PUT	/api/post/update/backend/{post_id}	Opdater tekniske data for Post	Implementeret, i brug	Virker
Post	4.4.5	DELETE	/api/post/remove/{post_id}	Slet Post	Implementeret, ikke i brug	Virker
Post	4.4.6	GET	/api/posts	Hent flere Post (filtrering understøttet)	Implementeret, i brug	Virker
Post	4.4.7	GET	/api/post/postsids?ids=1,2,3	Hent Post baseret på post-ID'er	Implementeret, i brug	Virker
Post	4.4.8	Post	/api/upload/image	Upload et billed	Implementeret, i brug	Virker
Interaktion	4.5.1	PUT	/api/post/vote/{post_id}	Stem på et Post (like/dislike/ingen)	Implementeret, i brug	Virker
Interaktion	4.5.2	GET	/api/post/{post_id}/vote/{user_id}	Hent stemmestatus for backend	Ikke implementeret	Ikke testet
Interaktion	4.5.3	POST	/api/post/comments/{post_id}	Opret kommentar til Post	Implementeret, i brug	Virker
Søgning	4.6.1	GET	/api/search/posts?q=	Søg i alle Post	Ikke implementeret	Ikke testet
Søgning	4.6.2	GET	/api/search/communities?q=	Søg i alle fællesskaber	Ikke implementeret	Ikke testet
Søgning	4.6.3	GET	/api/search/users?q=	Søg i alle brugere	Ikke implementeret	Ikke testet
Feedback	4.7.1	POST	/api/feedback/submit	Send feedback	Implementeret, i brug	Virker
Feedback	4.7.2	GET	/api/feedback/all	Hent al feedback	Implementeret, ikke i brug	Virker

Tabel 7: API endpoints

Som set i Tabel 7 er der enkelte endpoints som er implementeret men ikke gjort i brug, samt endpoints der endnu ikke er blevet implementeret da der alligevel ikke blev brug for dem.

9.2 Test 1 Fokusgruppen

Formål og Deltagergruppe

Test 1 foregik i form af en kvalitativ brugertest. Formålet med denne test var for at undersøge om applikationen var brugervenlig, om de forskellige funktioner fungerede som de skulle, samt forstå brugeres adfærd under brug af den. Testens deltagergruppe bestod af ni studerende fra Medialogi uddannelsen. De fleste personer der studerer på sådan en studieretning har en stor sandsynlighed for at høre under vores målgruppe, hvilket var årsaget til at de blev valgt.

Testen startede ud med en kort briefing hvor deltagerne i testen blev introduceret til applikationen. De fik også at vide hvilke funktioner der stadigvæk var under arbejde, at siden var en smule langsomt da det ikke var et optimeret produktions build.

Test Metode

Efter den korte briefing begyndte selve testen. Testen forløb med at deltagerne fik givet en bestemt opgave de skulle løse i applikationen. Dette kunne fx være at de skulle oprette et opslag. Når det var bekræftet at alle deltarere havde løst en opgave, fik de alle sammen givet den næste opgave, og sådan fortsatte det til at alle opgaverne var nået igennem. Udover disse opgaver, fik deltagerne intet at vide om hvordan applikationen fungerede, eller hvordan de skulle løse opgaven. På den måde kunne naturligt brugeradfærd bedre simuleres, og de mest nøjagtige resultater kunne indsamles. Listen af opgaver var som følgende.

- Lav en ny brugerprofil
- Opret et opslag
- Kommenter på et opslag lavet af en anden bruger
- Reager til et opslag (like eller dislike)
- Log ud, og log ind igen

Måden opgaverne var designet på, sikrede også at deltagerne ville teste de mest kritiske funktioner. Efter alle opgaverne var løst fik deltaerne også lov til frit at udforske hjemmesiden uden noget konkret mål. Formålet var her at observere brugeres adfærd når de ikke er styret af specifikke opgaver, hvilket bedre afspejler måden normale brugere ville agere. Derudover kunne det muligvis også afsløre problemer i dele af hjemmesiden som opgaverne ikke berørte.

Efter den praktiske brugertest, blev deltagerne spurgt om at svare på et spørgeskema. Det havde til formål at evaluere deltagernes oplevelse af applikationen. Der blev blandt andet spurgt ind til brugervenligheden af applikationen, om de oplevede nogle fejl, og hvor tilfredse de var med sidens funktionalitet. Der blev også indsamlet forslag til forbedringer og ønskede funktioner, og spurgt ind til hvor interesserede de egentlig var i applikationen. Spørgeskemaet havde en blanding af vurderingsskala spørgsmål og åbne spørgsmål. Vurderingsskalaerne blev brugt til at kvantificere holdninger og tilfredshed på en standardiseret måde hvor man hurtigt kan se generelle tendenser. De åbne spørgsmål gav mere kvalitative svar, hvor deltagerne fik mulighed for at uddybe deres svar, give specifik feedback, og komme med forslag til applikationen. Dette giver større indblik i hvorfor deltagerne svarede som det gjorde på skala-spørgsmålene.

Under testen blev nogle af deltagerne også bedt om at optage deres skærm under testen. Dette gav endnu en vektor at undersøge, og gav indsigt som ikke er muligt bare ved spørge-

skemaer. Der kan eksempelvis undersøges præcist hvordan brugeren navigere og interagere med hjemmesiden, hvor de sætter farten ned og hvor de løber ind i problemer. Dette er ting som en brugere sjældent husker hvis de fx svarer på et spørgeskema. Derudover kan de også sammenlignes med svarende i spørgeskemaet for bedre at forstå hvorfor der blev svaret som der gjordes.

Test Resultater

Efter testen blev svarende til spørgeskemaet analyseret for at finde frem til relevant indsigt. Overordnet så det ud til at delterne havde en nogenlunde positiv oplevelse, med de fleste bedømmelser liggende på 3-4 ud af 5. De fleste deltagere fandt siden nem at navigere, mens enkelte brugere oplevede mindre problemer. Designet blev modtagedet overvejende positivt, med nogle der beskrev det som "simpel", "ren" og "behageligt". Dog blev der også nævnt at det ikke var revolutionært, mens en anden synes det var "for meget", og var "lidt i overkanten". Det blev derfor konkluderet at meningen om designet var meget subjektivt, og det derfor skulle beholdes som det var. Dog lod det til at størstedelen mente at det var simpelt og intuitivt, hvilket var et af de vigtigste elementer. Ydermere var der også mange forslag til forbedringer og funktioner som deltagerne ønskede.

Ud fra dette blev der lavet en liste af ting der kunne implementeres. Listen kan ses i Tabel 8.

ID	Feature	Status
0	Omaranger knapper	Implementeret
1	Feedback	Implementeret
2	Post appending to community	Implementeret
3	User page	Implementeret
4	Community Join Button	Ikke Implementeret
5-1	Default to current community	Implementeret
5-2	Username + comname instead of id	Implementeret
5-3	Render lim	Implementeret
5-4	Comment feedback	Implementeret
6	Search	Ikke Implementeret
7	Filter (tags, tabs)	Ikke Implementeret
8	Billeder	Implementeret
9	User settings	Ikke Implementeret men brugt visuelt
10	Back buttons	Ikke Implementeret
11	Sign out button	Implementeret
12	Make bg - blur connection obvious	Ikke Implementeret
13	Discover	Ikke Implementeret
14	Security center info from about page	Ikke Implementeret

Tabel 8: Resultater af T1

[30] Nogle af tingende havde vi allerede tænkt os at ændre/tilføje.

9.3 Test 2 public test

Formål og Deltagergruppe

Test 2 blev udført da havde fået implementeret nok funktionalitet til at Gnuf var brugbart som socialt medie. Denne test bestod af at lancere hjemmesiden online og reklamere den på universitetet, i håb at at en rimelig mængde af personer havde lyst til at prøve det. Formålet var at observere mere naturligt adfærd en muligt under en kontrolleret test, indsamle generel feedback fra dem, samt at identificere fejl og potentielle exploits som vi ikke før havde opdaget.

Test Metode

Efter at hjemmesiden var bragt online, startede testen med at bringe opmærksomhed på Gnuf ved at hænge plakater rundt omkring på universitetet. Siden inkluderede en feedback side, hvor brugere kunne skrive feedback til siden. Udover dette, blev systemets performance og serverlogs også observeret, for at se om der her opstod nogle problemer.

For at styrke engagementet under testen og samtidig belønne dem, der bidrager positivt, blev der indført et præmiesystem. Det fungerede ikke som en lodtrækning eller tilfældig belønning, men handlede direkte om, hvor aktiv hver bruger var på platformen. Jo mere man poster, kommenterer og interagerer med andre, desto større er chancen for, at man får præmien. For at forhindre spam og andet uønsket adfærd, blev brugere som lavede spam opslag, eller på nogle måde exploitede applikationen. Til sidst blev brugeren med med mest aktivitet valgt som vinderen.

Resultater

Ud fra testen opdagede vi adskillede fejl og mangler der kunne adresseres. Udover det blev der også fundet to exploits. Den første exploit blev fundet af en bruger, som gjorde det muligt for dem at like/dislike posts på vegne af andre brugere. Dette var muliggjort grundet manglende autorisation for det endpoint, hvilket betød at man kunne spoofe sit bruger id, og derved troede systemet at man var en anden person. Det andet exploit blev fundet i gruppen på baggrund af den første exploit. Det blev opdaget at den samme fejl var til stede for endpointet til at slette brugere. Ved brug af denne exploit kunne hvilken som helst bruger slette andres profiler, hvilket er en meget uheldig situation. Begge exploits blev hurtigt midigeret ved at fjerne den fejl der muliggjorde dem, heldigvis inden den anden exploit blev fundet af nogle uden for udviklerholdet.

I forhold til hjemmesiden og serverens ydeevne, opstod der her ingen problemer.

ID	Status	Title	Description	Affected Component	Impact Scope
0	Fikset	Godkendelsesomgåelse via DevTools - Afstemning	Brugere kan omgå godkendelsestjek via browserens DevTools (F12) og stemme som en hvilken som helst bruger.	Backend	All Users
1	Fikset	Godkendelsesomgåelse via DevTools - Sletning af bruger	Brugere kan omgå godkendelsestjek via DevTools og slette vilkårlige brugerkonti.	Backend	All Users
2	Addressed	Chromium nedbrud via Om os-siden	Under visse forhold kan siden "Om os" få Chromium-browsere til at gå ned.	Client-Side	Individual User
3	Ikke fikset	Servernedbrud via Om os-siden	I meget sjældne tilfælde kan bestemte interaktioner på siden "Om os" få serveren til at gå ned.	Server-Side	All Users
4	Fikset	ESC-tast fungerer ikke ved oprettelse af opslag	Tryk på `Esc` lukker ikke oprettelsesvinduet som forventet.	Frontend	Individual User
5	Fikset	Problemer med visning af ASCII-kunst	ASCII-kunst vises forkert på grund af manglende linjeskift.	Frontend	All Users
6	Fikset	Feedback afvises ved vurdering 0	Indsendelse af feedback med vurdering 0 resulterer i en 400-fejl.	Frontend	All Users
7	Ikke fikset	Uendelig opdateringssløjfe for adgangstoken	Frontend forsøger gentagne gange at opdatere adgangstokens uden tilbageholdelse eller fejlretning, hvilket kan spamme backend og overfylde loggene.	Frontend	All Users
8	Ikke fikset	Godkendelsesfejl: Ugyldig Callback-URL	Frontend udløser ofte "InvalidCallbackUrl"-fejl, sandsynligvis pga. forkert konfigurerede eller uoverensstemmende OAuth-callback-URL'er, hvilket fører til over 5.000 gentagelser.	Frontend/Auth	All Users
9	Ikke fikset	Mangler WebRootPath	Servering af statiske filer virker ikke pga. ugyldig eller manglende WebRootPath, hvilket påvirker leveringen af frontend-assets.	Backend/Static File Middleware	All Users
10	Ikke fikset	Kommentarantal opdateres ikke	Efter indsendelse af en kommentar opdateres antallet ikke før man manuelt opdaterer siden, hvilket forvirrer brugeren.	Frontend/Comments	All Users
11	Fikset	Indstillingsside låser sessionen	Ændring af baggrundens gennemsigtighed låser indstillingsmenuen, og brugeren skal logge ind igen for at fortsætte.	Frontend/Settings	All Users
12	Fikset	Indstillinger gemmes ikke på tværs af sessioner	Brugerindstillinger gemmes eller gendannes ikke efter sessionsgenoprettelse, så præferencer som baggrundens gennemsigtighed nulstilles.	Frontend/Settings	All Users
13	Fikset	Log ud ved opslag viser ikke login-siden	Log ud mens man ser et opslag sender ikke brugeren tilbage til login-siden, men efterlader dem på en utilgængelig side.	Frontend/Auth Routing	All Users

Tabel 9: Resultater af T2

Problem ID 7 blev ikke fikset da det viste sig at problemet ikke var et problem da den ikke kunne finde brugerens adgangstoken (jwt-Token) når man ikke var logget ind, hvilket giver mening.

Problem ID 2 blev ikke fikset, da vi ikke har dyb nok forståelse af chromiums opbygning til at kunne addresere sådanne problem

Problem ID 13 var nemt at fikse da det viste sig at der ikke var givet en redirect path.

10 Diskussion

10.1 Designudvalg

Da vi begyndte at udvikle GNUF, var det vigtigt for os at skabe et design der både var funktionelt, brugervenligt og havde et udtryk som passede til målgruppen. Vi ville gerne lave noget der så godt ud, men som også føltes naturligt at bruge, især for de brugere der har interesse i teknologi og digitale fællesskaber. Det var vigtigt for os at siden ikke virkede overvældende eller fyldt med unødige elementer, men samtidig heller ikke kedelig eller gammeldags. Et af de første valg vi stod overfor var om vi skulle gå efter et meget visuelt og lidt mere moderne udtryk med animationer og effekter, eller holde os til noget simpelt og funktionelt. Vi endte med at vælge den enkle vej, dog med nogle effekter, som brugeren selv kunne vælge at bruge eller ej, såsom ændre baggrundsbillede eller baggrundsslør. Det gjorde vi fordi det er nemmere for brugerne at navigere i, og det giver brugeren mulighed for at personliggøre deres egne oplevelser.

Farvevalget var også noget vi brugte tid på at overveje. Vi ville gerne have noget der gav siden karakter, men som samtidig ikke blev for skrigende. Her faldt valget på sort og lilla som de gennemgående farver. Sort giver et rent og professionelt udtryk og fungerer godt som baggrund, især når man gerne vil undgå at øjnene bliver trætte efter længere tids brug. Lilla valgte vi fordi det giver lidt kant og identitet til siden. Den tilføjer noget liv uden at tage fokus væk fra indholdet. Vi var klar over at lilla måske ikke er en farve der appellerer til alle, men vi syntes den gav platformen et lidt mere kreativt og teknologisk præg.

Som tidligere nævnt Afsnit 9.2, blev en brugertest anvendt til at evaluere hvor godt sidens design var. Mange brugere fremhævede at siden var let at finde rundt i, og at funktionerne var logisk opbygget. De kunne nemt finde knapper og menuer og syntes generelt at det hele hang godt sammen. Vi fik også en del konkrete forslag til små forbedringer, som for eksempel justering af tekststørrelser og placering af visse funktioner, såsom tilbageknap eller at registreringsknappen var svær at finde. Nogle havde forslag til hvad der kunne gøre platformen endnu mere personlig eller intuitiv, og flere af dem har vi forsøgt at tage med videre i vores videreudvikling. Overordnet set bekræftede testen os i at vi havde ramt en retning der fungerer, men den mindede os også om at der altid vil være forskelle i hvordan folk oplever design, og at man sjældent kan lave noget der passer perfekt til alle.

Noget af det vi har taget med os fra processen er vigtigheden af at teste løbende og ikke vente til alt er færdigt. Det giver mulighed for at rette til og forbedre inden noget bliver låst fast. Til sidst kan man sige at hele designprocessen har handlet om at finde balancen mellem det tekniske og det visuelle, mellem det enkle og det karakterfulde. Vi har ikke prøvet at opfinde den dybe tallerken, men vi har forsøgt at skabe en platform der føles velkendt, men stadig har sit eget præg og som folk har lyst til at bruge.

10.2 I forhold til andre platforme

En af de prioteret valgt i vores projekt var at gøre platformen målrettet teknologientusiaster. Det betyder, at vi har valgt udelukkende at fokusere på IT-relaterede emner, i stedet for at lave en bred platform med alt fra livsstil til politik. Vi har struktureret siden omkring forskellige communities, som hver især dækker specifikke emner. Tanken er, at brugerne skal kunne finde og deltage i præcist de samtaler, der matcher deres interesser, uden at blive distraheret af alt muligt andet irrelevant indhold.

Denne måde at strukturere indhold på er blandt andet inspireret af Reddit, som vi kigget på i vores State of Art afsnit. Det understreger, hvor vigtigt det er at lytte bredt og skabe balance i brugerinddragelsen, så man undgår at give for meget magt til en lille gruppe, som ikke nødvendigvis repræsenterer helheden. Her skal man tage hensyn til alle brugere på hjemmesiden. Derfor har vi i vores projekt valgt en anden tilgang i form af det primæresystem tidligere beskrevet i Afsnit 9.3. Pointen er, at vi gerne vil motivere folk til at deltage på en måde, der gør fællesskabet bedre. Ikke ved at konkurrere om mest opmærksomhed, men ved at være hjælpsomme, dele viden og skabe et godt miljø for andre. I modsætning til Reddit, hvor opmærksomhed ofte kan dominere, forsøger vi at fremhæve kvalitet og indsats. Det har vi prøvet at opnå ved at lave et belønningssystem, hvor præmien fungerer som en form for anerkendelse, der kan være særlig vigtig for nye brugere. I stedet for at skulle kæmpe om opmærksomhed eller have mange følgere, kan man blive belønnet for bare at være aktiv og konstruktiv. Det er med til at gøre platformen mere åben og inkluderende for alle. Sammenlignet med Reddit prøver vi altså at skabe en mere fokuseret og brugerstyret platform, hvor teknisk viden og fællesskab er i centrum. Ved at kombinere emnespecifikke communities, erfaringer fra eksisterende platforme og et system der belønner god deltagelse, tror vi på, at vi kan tiltrække og fastholde en målgruppe, som ofte savner netop det i andre sociale medier.

10.3 Valg af frontend teknologi

Under udviklingen af vores løsning var der nogle aspekter af vores udvikling af løsning der var forudbestemt, et af dem var at vores backend skulle skrives i C# og at vores system skulle indebære *Complex Data Management*. Med dette grundlag havde vi flere veje, vi kunne gå med at udvikle og designe det endelige system.

Et aspekt som vi gjorde os flere overvejelser om var brugergrænsefladen (frontend), som spiller en central rolle i et socialt medie, da brugeroplevelsen har stor betydning for, hvordan brugerne interagerer med platformen. Derfor har vi lagt vægt på at vælge teknologier, der understøtter en moderne, responsiv og brugervenlig frontend. Frontend-delen af vores løsning er udviklet med fokus på stabilitet, genbrugelighed og brugeroplevelse. Vi har anvendt moderne teknologier som React, TypeScript og Tailwind CSS. Desuden har vi sikret en klar adskillelse mellem frontend og backend (.NET API), hvilket muliggør uafhængig udvikling og test.

Når man bygger et større projekt med mange komponenter kan projektet hurtigt blive komplekst og svært at overskue uden ordentlig strukturering af programmeringen. Med Reacts komponent baseret tilgang gøres dette nemmere. Selvom React, TypeScript og Tailwind er en stærk kombination, kræver det en del viden og tid at lære, især for nybegyndere. Dette kan dermed gøre det sværere at komme hurtigt i gang sammenlignet med mere simple løsninger

med HTML/CSS og Javascript. Men vi stod heldigvis med nogle gruppemedlemmer, der havde kendskab til React og TypeScript, hvilket gjorde denne proces en del nemmere. Da vi bruger C#/.NET i vores backend er vores frontend og backend skrevet i to forskellige teknologier, hvilket ikke sikrer en fuld integration. Dette kræver ekstra arbejde, da de ikke forstår hinanden og dermed kommunikere vi igennem API'er (HTTP Kald). Vi skal derfor sørge for at data sendes korrekt frem og tilbage, og det kan skabe fejl, hvis dataformater ikke stemmer overens. I starten af udviklingsprocessen fik vi udarbejdet database dokumentation, som tydeliggjorde hvilke data der findes i systemet, hvordan data hænger sammen og hvordan frontend og backend skal læse/skrive data korrekt. Med en mere præcis API var integrationen nemmere og mindskede fejl. Det gjorde det også nemmere at vedligeholde, da ændringer i databasen medførte opdatering af dokumentation, så alle hurtigt kunne se hvad der skulle ændres.

Et alternativ til vores frontend-løsning kunne have været at bruge Blazor, som er en webudviklingsteknologi fra Microsoft, hvor man bygger frontend med C# i stedet for JavaScript eller TypeScript. Blazor fungerer i browseren gennem WebAssembly eller som en serverbaseret løsning, og det giver mulighed for at bygge interaktive webapps med samme sprog og framework som i backend (.NET og C#). Hvis vi havde brugt Blazor, kunne vi have skrevet hele systemet – både frontend og backend – i C#, og dermed haft fælles datamodeller, valideringsregler og forretningslogik. Det ville betyde, at vi kunne genbruge mere kode og undgå fejl, som kan opstå, når man skal oversætte data mellem to forskellige sprog (TypeScript og C#). Det kunne også have gjort integrationen lettere, fordi frontend og backend ville være fuldt integreret under samme .NET-platform. Konkret kunne vi med Blazor have bygget brugergrænsefladen direkte i Visual Studio, og ladet komponenter som opslag, likes og kommentarer fungere som C#-komponenter, i stedet for React-komponenter. Vi ville stadig skulle forbinde dem med databasen og API'er, men det ville alt sammen foregå inden for samme teknologiske ramme.

Endnu et alternativ ville havde været at bruge Next.js som et fullstack framework, i stedet for blot frontend. Dette ville give de samme fordele som Blazor, og samtidig med at beholde de fordele der kommer med React og Next.js. Dette var desvære ikke en mulighed grundet kravet om en backend skrevet i C#, ellers havde dette været den valgte tilgang.

Selvom det betød en seperat backend, valgte vi dog stadig React og TypeScript, fordi vi vurderede, at det gav større fleksibilitet og en bedre brugeroplevelse, samt fordi vi havde erfaring med disse teknologier. Men Blazor kunne have været en god løsning, hvis vi ønskede tættere integration, mindre kompleksitet og færre teknologier at vedligeholde, og en fuld Next.js løsning havde været optimal hvis vi havde haft mulighed for det.

10.4 Sikkerhed

Sikkerhed har spillet en vigtig rolle i udviklingen af vores løsning, især fordi platformen håndterer brugerdata og login-funktionalitet. Vi har sørget for, at adgangskoder ikke gemmes i klartekst, men i stedet bliver hash'et, så de ikke kan læses selv hvis databasen kompromitteres. Dette er en vigtig grundregel i sikkerhed og bidrager til beskyttelse af brugernes privatliv.

Da frontend og backend er bygget i forskellige teknologier, har vi også lagt vægt på at validere data i begge ender for at forhindre fejl og potentielle angreb som fx. SQL injections([31]). Vi har struktureret vores database og API'er, så kun nødvendige data udveksles, og vi har tænkt anonymitet ind i designet ved ikke at gemme personfølsomme oplysninger.

Selvom vi har taget flere grundlæggende sikkerhedsforanstaltninger, kunne systemet i en fremtidig version forbedres med krypteret kommunikation (HTTPS) til backend, rollebaseret adgangsstyring, og en mere komplet autentificeringsløsning([32]). Derudover kunne vi arbejde med GDPR-hensyn, så brugeren får mere kontrol over sine egne data.

10.5 Metode udvalg

Under udviklingen af vores løsning arbejdede vi ud fra en agil tilgang inspireret af Scrum, dog i en forenklet form, tilpasset vores gruppe og projektets omfang. Vi havde ikke faste roller som Scrum Master eller Product Owner, men vi arbejdede iterativt, hvor opgaver blev planlagt, fordelt og løst i mindre faser. Vi brugte værktøjer som Github,Dicord og Typst til at holde styr på vores opgaver og fremdrift, og vi holdt uformelle statusmøder, hvor vi løbende vurderede, hvad der fungerede, og hvad der skulle justeres.

Denne tilgang gav os en fin grundstruktur, hvor vi kunne arbejde fokuseret og målrettet i korte forløb lidt i stil med sprints, men vi havde dog ikke en fastlagt sprint. Det gav os mulighed for at opbygge funktionalitet i mindre bidder og samtidig have plads til at reagere på ændringer, især efter brugertests, hvor vi modtog feedback, der krævede justeringer i UI'et og funktionaliteten. Dog var vores tilgang ikke uden udfordringer. Scrum forudsætter ofte en fast rytme, klare sprintmål og en høj grad af planlægning. I praksis arbejdede vi mere fleksibelt og ikke altid i faste iterationsforløb, og nogle gange blev opgaver løst ad hoc. Her kunne en anden agil metode som Kanban muligvis have været mere velegnet til vores arbejdsproces.

Kanban, som er en agil metode udviklet af toyota. Kanban fokuserer ikke på iterationer, men i stedet på at visualisere arbejdet og styre flowet af opgaver gennem kolonner som "To Do", "In Progress" og "Done"([33]). Det kunne have passet bedre til vores behov for løbende tilpasninger, især i faser hvor opgaver og prioriteter hurtigt ændrede sig f.eks. når integrationen mellem frontend og backend krævede tekniske justeringer, eller når vi mødte uventede problemer med deployment. Med Kanban kunne vi have haft større fleksibilitet og et mere realistisk overblik over, hvor vi faktisk befandt os i processen uden at være bundet af sprintmål.

Omvendt havde Kanban måske gjort det sværere for os at skabe tydelige delmål og holde fokus, særligt op mod afleveringsfrister. Scrum giver en mere tydelig målstyring og gør det nemmere at sikre fremdrift i de faser, hvor opgaverne var mere komplekse og krævede koordinering især i forhold til design af datamodeller, API-struktur og frontend funktionalitet.

Alt i alt vurderer vi, at vores valgte tilgang fungerede, men skal forbedres. Vi fik kombineret struktur med fleksibilitet. Det vigtigste var ikke at følge en metode slavisk, men at kunne

tilpasse os vores projekt, vores gruppe og de tekniske udfordringer vi mødte. Hvis vi skulle have arbejdet videre i en længere periode, ville vi dog overveje at eksperimentere med andre metoder, såsom Kanban, der kunne støtte op om dynamisk arbejde med mere struktur.

10.6 SQLite vs PostGreSQL

Valget af et passende databasesystem var en vigtig beslutning når det kom til projektets arkitektur. Det har stor indflydelse på projektets ydeevne, skalerbarhed, samt hvilke funktioner der er til rådighed. I GNUF's tilfælde var hardwaren en stor flaskehals, eftersom at databasen skulle være i stand til at køre på en Raspberry Pi. Derfor blev SQLite overvejet som en mulighed grundet dets letvægtighed. På den anden side ville et socialt medie have komplekse krav når det kommer til database, hvor SQLite's mangel på funktioner kunne vise sig at være et problem. Af denne årsag blev PostGreSQL også overvejet som en mulighed grundet dets robusthed og omfattende funktionalitet.

PostGreSQL

PostGreSQL er et relationelt databasesystem med mange funktioner velegnet til et socialt medie som GNUF. PostGreSQL tilbyder blandt andet mere avancerede datatyper og indeksering, som giver mulighed for mere komplekse datastrukturer [34]. Dette inkluderer blandt andet JSON datatyper og arrays, hvilket er noget SQLite mangler. PostGreSQL tilbyder også betydeligt bedre skalerbarhed end SQLite, hvilket ville være en markant fordel hvis der skulle opstå en stor stigning i brugen af GNUF. En anden fordel er PostGreSQL's rollebaserede adgangssystem som forbedrer applikationens sikkerhed. Dette system muliggører oprettelsen af specifikke brugerroler med definerede tilladelser, og giver præcis kontrol over hvad forskellige brugere har rettigheder til at gøre.

Derudover er PostGreSQL ACID-compliant, hvilket betyder at det sikrer datatransaktioner gennem fire egenskaber [35].

- "Atomicy", som garanterer at for hver transaktion med flere trin, fuldføres enten alle trin succesfuldt, eller også træder ingen af dem i kraft.
- "Consistency" sørger for at databasen forbliver i en valid tilstand før og efter transaktioner.
- "Isolation" forhindrer samtidige transaktioner i at forstyrre hinanden, hvilket er vigtigt i systemer som tillader flere brugere samtidig.
- "Durability" bekræfter at afsluttede transaktioner forbliver permanente, også under system fejl.

På trods af disse mange styrker, gav PostGreSQL anledning til udfordringer i forbindelse vores Raspberry Pi's begrænsede ydeevne. Det største problem var PostgresSQL større ressourcekrav til hukommelse og processorkraft, hvilket muligvis kunne putte serveren under pres. Udover det er PostGreSQL også mere kompliseret at implementere i forhold til SQLite, dog havde dette mindre indflydelse på hvilket system der skulle anvendes.

SQLite

SQLite er en letvægts, self-contained, serverless database. I modsætningen til andre database systemer som ***PostGreSQL*** eller MySQL, kører SQLite ikke som en separat process, men eksistere bare lokalt på disken, og bliver tilgået direkte af backenden. Dette minimere overhead, og gør denne ideel til små indlejrede systemer. [36]

I GNUF projektet, som kører fuldstændigt på et letvægts indlejret system (se Afsnit 8.4) tilbyder SQLite præcis hvad vi har brug for;

- Minimal configuration for hurtigere udvikling og testning
- Høj ydeevne, særligt når filen caches i RAM hvilklet tillader super low latency adgang til filen
- En database som ikke kræver endnu et system som skal bootes og administreres via Systemd (se Afsnit 8.4) og kræver RAM, CPU cycles, etc.

Dog kommer disse fordele ikke uden deres konsekvenser. SQLite er ikke særlig god til samtidige skrivninger. Da det blot er en enkelt fil, kan flere ting sagtens læse på samme tid, men for at undgå race konditioner, bør maks 1 ting skrive af gangen. Dette *kan* lede til et meget langsomt system, *hvis* flere processer og/eller noder har brug for at tilgå filen på en gang. Dette er dog ikke tilfældet her da vi kun har en node, med en backend. Og mens flere endpoints måske vil skrive på samme tid, er dette heller ikke et problem da

- BE skriver async, så venter ikke på at databasen bliver skrevet før vi kan fortsætte udførsel.
- GNUF har en meget lille forventet brugerbase, på mindre end 50 brugere, derfor forventes det ikke at mange samtidige skrivninger kommer til at opstå

SQLite har heller ingen indbygget adgangskontrol, hvilket hurtigt kan blive en sikkerheds risiko. Da dette er et proof of concept på en Raspberry PI kun vi har adgang til, er dette ikke den største bekymring, men hvis vi gik til produktion, skulle klart vælges noget mere sikkert end SQLite. Dens manglende adgangskontrol er dog også en fordel, da denne gør databasen nem kopierbar, samt analyserbar. [36] Til slut kan SQLite ikke rigtigt scales, primært grundet dens manglende evne til at håndtere samtidige skrivninger. Dette vil sige at hvis GNUF voksede, og vi havde brug for mere end en server til at håndtere brugerbasen (f.eks via et kubernetes cluster eller lign), ville eneste måde at få det til at fungere på med SQLite være at have en kopi af databasen på hver server, og have en separat process til at synkronisere x antal gange om dagen (hvor ingen andre kan skrive til db imens), hvilket ikke er en optimal løsning.

Beslutningen om databasen

Valget mellem PostGreSQL og SQLite blev i sidste ende truffet med udgangspunkt i GNUF's faktiske krav og kontekst. Hvor PostGreSQL tilbyder en lang række avancerede funktioner, stærk skalerbarhed og robust adgangskontrol, medfører det også et væsentligt større ressourceforbrug og højere kompleksitet – noget der ikke matcher den platform GNUF er designet til at køre på.

GNUF er et proof-of-concept bygget til en enkelt Raspberry Pi med en begrænset og kendt brugerbase. Under disse forhold er SQLite et oplagt valg. Det tilbyder en letvægtsløsning med minimal konfiguration, lavt ressourcetræk og høj ydelse – særligt når databasen caches i RAM. Samtidige skrivninger er ikke et praktisk problem i vores arkitektur, da vi har en enkelt backend-instans og en forventet lav mængde samtidige transaktioner.

Selvom SQLite har begrænsninger hvad angår skalerbarhed og adgangskontrol, er disse ikke relevante i GNUF's nuværende form. Derimod er enkelheden og driftssikkerheden ved en self-contained database, der ikke kræver en separat serverproces, et stort aktiv i et miljø med begrænsede ressourcer.

På den baggrund vurderes SQLite til at være det bedst egnede valg for GNUF, set i forhold til både systemarkitektur, udviklingshastighed og projektets realistiske krav.

Funktion	PostgreSQL	SQLite
Funktionalitet	Avanceret	Enkel
Skalerbarhed	Meget høj	Lav
Brugsscenarie	Store systemer, produktion	Prototyper, små apps
Ydeevne ved mange samtidige brugere	God	Begrænset
Ressourceforbrug	Højt	Lavt

Tabel 10: SQLite vs PostGreSQL: fordele og ulemper

11 Konklusion

Med den tidsbegrænsning projektet har varet har vi prøvet at gøre hjemmesiden så sikker som muligt, eksempelvis ved at teste systemet i forhold til kravspecifikationerne, ved at lave hashed passwords, uden at gemme data om brugeren, medmindre brugeren selv har valgt at indsætte disse data osv. Brugeren kan vælge at interagere med hjemmesiden anonymt uden behov for at indsætte navn eller andre personfølsomme informationer, dermed gøre det mere trygt at navigere hjemmesiden, skulle brugeren vælge ikke at dele eksempelvis navn eller e-mail. Brugernes anonymitet og datasikkerhed er blevet prioriteret i forhold til at brugerne selv vælger hvilke data de deler, dermed give autonomi til brugerne når de færdes rundt i GNUF platformen.

Brugertestene har givet en overordnet overblik over målgruppen meninger om platformen, som har vist at det er målrettet til IT enthusiaster. Dette er blevet gjort ved at udføre disse tests på forskellige målgrupper, hvor langt de fleste af brugerbasen er CCT studerende, samtlige communities er relateret til IT. Målgruppen som ikke har været CCT, har ikke følt at mediet er tilpasset dem, dog ser potentiale ved platformen trods den manglende indhold på tidspunktet hvor brugertests blev udført. Dette har medført at målgruppen har været mere tilbageholdende fra at interagere med siden. Det visuelle design er blevet bekræftet behageligt og intuitivt at navigere rundt i gennem brugertests.

Databasens organiserede struktur er blevet opnået ved at udarbejde dokumentation for databasens opsætning og benytte SQLite, da dens ressourceforbrug og høje ydeevne har gjort datahåndteringen effektiv. Kommunikationen mellem databasens tabeller og backend udviklingens controllere og modeller har givet projektet mere overblik over hvordan data håndteres.

I henhold til GDPR har det blandt andre kun været muligt at overholde dele af artikkel 7, da vi ikke har muligheden for at slette brugere eller deres data på anmodning. Al data bliver selvfølgelig slettet ved projektets afslutning ved en formatering af disken som serveren kører på. Dog har vi taget foranstaltninger for at overholde GDPR mest muligt. Blandt andet har vi en boks hvor folk skal verificere, at de er over 16 for at de må oprette en konto. Dette gøres for at undgå konflikt med GDPR art. 8.

Images

Tables

Kode Blok

File Tree

Bibliografi

[1] geeksforgeeks, "introduction to unix system". [Online]. Tilgængelig hos: https://www.geeksforgeeks.org/introduction-to-unix-system/

[2] GitHub, "Typst". [Online]. Tilgængelig hos: https://github.com/typst/typst

[3] Github, "Github". [Online]. Tilgængelig hos: https://docs.github.com/en/get-started/start-your-journey/about-github-and-git

[4] Discord, "Discord". [Online]. Tilgængelig hos: https://discord.com/

[5] Figma, "Figma". [Online]. Tilgængelig hos: https://www.figma.com/design/

[6] M. Danielsen, "Hvad er sociale medier?", *Socialemedier.dk*, 2015, [Online]. Tilgængelig hos: https://www.socialemedier.dk/hvad-er-sociale-medier/

[7] L. Fangfang, J. Larimo, og L. C. Leonidou, "Social media marketing strategy: definition, conceptualization, taxonomy, validation, and future agenda", *Springer Nature*, 2020, [Online]. Tilgængelig hos: https://link.springer.com/article/10.1007/s11747-020-00733-3

[8] C. V. Baccarella, "Social media? It's serious! Understanding the dark side of social media", *Science Direct*, 2018, [Online]. Tilgængelig hos: https://www.sciencedirect.com/science/article/pii/S0263237318300781

[9] AIContentfy-team, "Behind the Scenes: Building a Social Media Platform like Facebook", 2024, [Online]. Tilgængelig hos: https://aicontentfy.com/en/blog/behind-scenes-building-social-media-platform-like-facebook

[10] T. Swartz og E. Hamilton, "What 5 years at Reddit taught us about building for a highly opinionated user base", *Lenny's Newsletter*, 2023, [Online]. Tilgængelig hos: https://www.lennysnewsletter.com/p/what-5-years-at-reddit-taught-us

[11] J. N. Matias, "What Just Happened on Reddit? Understanding The Moderator Blackout", *Social Media Collective*, 2015, [Online]. Tilgængelig hos: https://socialmediacollective.org/2015/07/09/what-just-happened-on-reddit-understanding-the-moderator-blackout/

[12] R. Dolan, J. Conduit, C. Frethey-Bentham, J. Fahy, og S. Goodman, "Social media engagement behavior: A framework for engaging customers through social media content". [Online]. Tilgængelig hos: https://kbdk-aub.primo.exlibrisgroup.com/discovery/fulldisplay?docid=cdi_webofscience_primary_000487035200009CitationCount&context=PC&vid=45KBDK_AUB:AUB&lang=da&search_scope=MyInst_and_CI&adaptor=Primo%20Central&tab=Everything&query=any,contains,Social%20media%20engagement%20behavior:%20A%20framework%20for%20engaging%20customers%20through%20social%20media%20content&offset=0&searchInFulltext=false

[13] M. Sichach, "Uses and Gratifications theory - Background, History and Limitations", *SSRN*, 2023, [Online]. Tilgængelig hos: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4729248#:~:text=Abstract,integrative%2C%20and%20tension%2Dfree.

[14] M. Trunfio og S. Rossi, "Conceptualising and measuring social media engagement: A systematic literature review", *Italian Journal of Marketing*, bd. 2021, s. 267–292, 2021, [Online]. Tilgængelig hos: https://www.researchgate.net/publication/354951823_Conceptualising_and_measuring_social_media_engagement_A_systematic_literature_review

[15] S. Trepte, "The social media privacy model: Privacy and communication in the light of social media affordances". [Online]. Tilgængelig hos: https://www.researchgate.net/profile/Sabine-Trepte/publication/341231711_The_Social_Media_Privacy_Model_Privacy_and_Communication_in_the_Light_of_Social_Media_Affordances/links/605e1264458515e83472de48/The-Social-Media-Privacy-Model-Privacy-and-Communication-in-the-Light-of-Social-Media-Affordances.pdf?origin=publication_detail&_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6Il9kaXJlY3QiLCJwYWdlIjoicHVibGljYXRpb25Eb3dubG9hZCIsInByZXZpb3VzUGFnZSI6InB1YmxpY2F0aW9uIn19

[16] A. Acker og A. Kreisberg, "Social media data archives in an API-driven world", *Archical Science*, s. 106–120, 2019, [Online]. Tilgængelig hos: https://www.proquest.com/docview/2296470428/fulltextPDF?accountid=8144&parentSessionId=hItbpiS6ONPAHvkS1iqwJlB5QgRRbBHecFlRP8CwyCQ%3D&pq-origsite=primo&sourcetype=Scholarly%20Journals

[17] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, og J. M. Patel, "SQLite: Past, Present, and Future". [Online]. Tilgængelig hos: https://www.vldb.org/pvldb/vol15/p3535-gaffney.pdf

[18] Meta og the React community, "React API Reference". [Online]. Tilgængelig hos: https://react.dev/reference/react

[19] Next.js, "What is React and Next.js". Set: 14. maj 2025. [Online]. Tilgængelig hos: https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs

[20] T. Labs, "Tailwind CSS". [Online]. Tilgængelig hos: https://tailwindcss.com/

[21] shadcn, "shadcn/ui Documentation". [Online]. Tilgængelig hos: https://ui.shadcn.com/docs

[22] Vercel, "Routing". [Online]. Tilgængelig hos: https://nextjs.org/docs/pages/building-your-application/routing

[23] Auth.js, "Session Strategies". [Online]. Tilgængelig hos: https://authjs.dev/concepts/session-strategies

[24] Z. Team, "Zod". [Online]. Tilgængelig hos: https://zod.dev/

[25] "Components/Dialog". [Online]. Tilgængelig hos: https://ui.shadcn.com/docs/components/dialog

[26] Microsoft, “ASP DotNet Core documentation”. [Online]. Tilgængelig hos: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0&WT.mc_id=dotnet-35129-website

[27] gnuf, “API Specsheet”. [Online]. Tilgængelig hos: https://github.com/CCT2P2/Report/blob/main/spec_sheets/DB/API_SPECSHEET.md

[28] OWASP, “Password storage cheat sheet”. [Online]. Tilgængelig hos: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[29] A. Wiki, “systemd - ArchWiki”. [Online]. Tilgængelig hos: https://wiki.archlinux.org/title/Systemd

[30] gnuf, “T1 resultater”. [Online]. Tilgængelig hos: https://docs.google.com/spreadsheets/d/1p-nkXa38HTc89dnuZdNIdH4hdn3jnvQJYH3nAghBN1o/edit?usp=sharing

[31] crowdstrike, “injection attacks”. [Online]. Tilgængelig hos: https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/injection-attack/

[32] lrswebsolutions.com, “11 best practices for developing secure web application”. [Online]. Tilgængelig hos: https://www.lrswebsolutions.com/Blog/Posts/32/Website-Security/11-Best-Practices-for-Developing-Secure-Web-Applications/blog-post/

[33] asana.com, “what is kanban”. [Online]. Tilgængelig hos: https://asana.com/resources/what-is-kanban

[34] P. G. D. Group, “PostgreSQL Documentation”. Set: 19. maj 2025. [Online]. Tilgængelig hos: https://www.postgresql.org/docs/current/

[35] G. Vossen, “ACID Properties”, i *Encyclopedia of Database Systems*, L. LIU og M. T. ÖZSU, Red., Boston, MA: Springer US, 2009, s. 19–21. doi: 10.1007/978-0-387-39940-9_831.

[36] D. R. Hipp, “SQLite Documentation”. [Online]. Tilgængelig hos: https://www.sqlite.org/docs.html

[37] E. Union, “GDPR”. [Online]. Tilgængelig hos: https://gdpr_info.eu/

[38] T. P. G. D. Group, “PostgreSQL 17.4 Documentation”. [Online]. Tilgængelig hos: https://www.postgresql.org/files/documentation/pdf/17/postgresql-17-A4.pdf

[39] P. G. D. Group, [Online]. Tilgængelig hos: https://www.postgresql.org/

[40] Meta og the React community, “Build a React App from Scratch”. Set: 24. marts 2025. [Online]. Tilgængelig hos: https://react.dev/learn/build-a-react-app-from-scratch

[41] M. Trevisan, F. Soro, M. Mellia, I. Drago, og R. Morla, “Attacking DoH and ECH: Does Server Name Encryption Protect Users’ Privacy?”. [Online]. Tilgængelig hos: https://dl-acm-org.zorac.aub.aau.dk/doi/10.1145/3570726

[42] P. G. D. Group, “High Availability, Load Balancing, and Replication”. Set: 19. maj 2025. [Online]. Tilgængelig hos: https://www.postgresql.org/docs/current/high-availability.html

12 Bilag

Column name	Type	Constraints	Description
ID	INT	PRIMARY KEY	Unique Identifier
NAME	TEXT	UNIQUE. NOT NULL. CHECK (LENGTH(NAME) < 100)	Community Name
Description	TEXT	CHECK (LENGTH(description) < 500)	Community description
IMG_PATH	TEXT		URL to community image
MEMBER_COUNT	INT	DEFAULT 0. NOT NULL	Number of members
TAGS	str[csv]	NOT NULL	Content tags
POST_IDs	str[csv]		Array of post ID's (FK to posts.id)

Tabel 11: community stuckture

Column Name	Type	Constraints	Description
POST_ID	INT	UNIQUE. NOT NULL	Unique Identifier
TITLE	TEXT	NOT NULL. CHECK (LENGHT < 100)	Post title
MAIN_TEXT	TEXT	CHECK (LENGTH < 10k)	Post body
AUTH_ID	INT	NOT NULL	Author ID (FK. User.ID)
COM_ID	INT	NOT NULL	Community ID (FK. Community.ID)
TIMESTAMP	INT	NOT NULL	Time of post
LIKES	INT	NOT NULL. DEFAULT 0	Likes on post
DISLIKES	INT	NOT NULL. DEFAULT 0	Dislikes on post
POST_ID_REF	INT		Refference to original post (for reposts) (FK. Post.ID)
COMMENT_FLAG	BOOL	NOT NULL	Indicates comment instead of post
COMMENT_CNT	INT	NOT NULL. DEFAULT 0	Comment count
Comments	str[csv]		Array of post id's for comments (FK. Post.ID)

Tabel 12: post stuckture

Column name	Type	Constraints	Description
id	INT	PRIMARY KEY AUTO_INCREMENT	Unique identifier for each feedback entry
worked	STRING	NOT NULL	what worked
didnt	STRING	NOT NULL	what didnt work
feedback	STRING		additional feedback
rating	INT	NOT NULL	rating from 1 to 5
timestamp	INT	NOT NULL	timestamp of feedback entry

Tabel 13: feedback stuckture

Procesanalyse

Denne procesanalyse bygger på vores erfaringer fra projekt P2 og har til formål at skabe et mere effektivt og velfungerende grundlag for projekt P3. Analysen samler refleksioner over, hvad der fungerede godt i projektet, hvilke udfordringer vi mødte, og hvordan vi vil forbedre os fremadrettet.

Arbejdsform og koordinering

I P2 var det tydeligt, at vi manglede en egentlig struktur for, hvordan arbejdet skulle fordeles og planlægges. Der blev arbejdet fragmenteret, og det var ofte uklart, hvor langt vi var i processen. Vi havde nogle overordnede deadlines, men ingen konkrete delopgaver eller milepæle, der kunne guide arbejdet i hverdagen. Det gjorde det svært at fordele ansvar og skabe overblik. Selvom vi i P2 forsøgte at arbejde med en sprint-baseret tilgang, oplevede vi udfordringer med at håndhæve de aftalte deadlines. Ofte blev de deadlines, vi satte, ikke fulgt konsekvent, og de var heller ikke altid konkrete nok til at skabe en klar fælles retning. Det betød, at arbejdet nogle gange blev udskudt, og at det var svært at vurdere, hvornår en opgave egentlig var færdig. Til P3 vil vi derfor gå over til en mere struktureret og konsekvent sprint-baseret arbejdsform. Det betyder, at vi opdeler projektet i mindre perioder, fx af en uges varighed, hvor vi har klare mål. Hver sprint afsluttes med en intern evaluering, hvor vi vurderer, hvad der gik godt, og hvad der skal justeres. Vi vil samtidig arbejde med mere præcise og målbare deadlines og sikre, at der er en fælles forståelse af, hvad der skal være færdigt hvornår. Derudover vil vi være mere konsekvente i at følge op på de mål, vi sætter i starten af hver sprint. På den måde sikrer vi, at der hele tiden er progression, og at eventuelle problemer opdages i tide.

Et andet forslag til arbejdsstruktur er Trunk-Based development. Trunk-Based development er en måde at samarbejde om udvikling i Git på ved at opdele et projekt i små funktioner, oprette Kortlivede grene (typisk kun 1-3 dage) til en specifik funktion eller et patch. Hver gren flettes kun når en funktion er testet og fungerer. Dette fremskynder udviklingen, reducerer flettekonflikter og sikrer, at der altid er en fungerende demo eller et produkt, hvilket minimerer kodningssessioner i sidste øjeblik.

Teknisk engagement og faglig involvering

Et af de store problemer i P2 var, at ikke alle var lige involverede i kodningen. Det betød, at viden og ansvar blev samlet hos få personer, mens andre havde mindre indflydelse på de tekniske beslutninger. Det skabte en ubalance og gjorde det sværere at få et samlet overblik over projektets tekniske del. I P3 vil vi have fokus på at alle deltager aktivt i kodningen. Det gøres blandt andet ved at fælles kode(især i starten). Hvis alle er med fra starten og forstår den kode, der bliver skrevet, kan vi hurtigere opdage fejl, og det bliver lettere at tage over for hinanden, hvis nogen er fraværende. Vi vil også prioritere faglig diskussion og refleksion, så vi ikke bare koder løs, men også forstår, hvorfor vi gør det, vi gør. Det kan eksempelvis være korte sessioner, hvor vi præsenterer og forklarer vores tilgang til resten af gruppen. Vi udvalgte et projekt med en vis kompleksitet, hvilket satte store krav på den viden, der skulle opnås for at deltage aktivt i diskussionen. Vi vil derfor sætte større fokus på at udvælge et projekt, som er mere passende med det generelle niveau. Så alle kan lære, og stadig føle sig udfordret.

I starten af projektet havde vi en fragmenteret opstart på udviklingsdelen. I stedet for at samle os om en fælles kodebase og etablere fælles retningslinjer, begyndte flere på individuelle dele uden en koordineret tilgang. Vi blev først opdelt i tre mindre grupper med hver deres ansvarsområde, men det arbejde, vi lavede, var i praksis stadig tæt forbundet og afhængigt af hinandens fremdrift. Det gjorde det svært at arbejde selvstændigt, og små forsinkelser eller misforståelser i én gruppe kunne hurtigt påvirke de andre. Der blev heller ikke afsat tid til fælles introduktion eller onboarding til kodebasen, hvilket gjorde det svært for især nye eller mindre erfarne medlemmer at følge med. Det

er sandsynligt, at denne ustrukturerede opstart var medvirkende til, at omkring halvdelen af de teknisk ansvarlige gradvist faldt fra. Når man ikke føler sig inddraget eller har den nødvendige indsigt i projektet, er det nemt at miste motivationen. Til P3 vil vi derfor prioritere en mere struktureret og fælles opstart, hvor alle bliver grundigt introduceret til kodebasen og de værktøjer, vi bruger, så alle føler sig klædt på til at bidrage aktivt.

Kommunikation og fælles forståelse

I P2 var kommunikationen generelt god, men der var også eksempler på, at vi ikke fik justeret vores forventninger eller delt ændringer i tide. Når der blev lavet ændringer i specifikationen, blev det ikke altid meldt klart ud. Det skabte forvirring og tidsspild. Derfor vil vi i P3 arbejde mere systematisk med versionsstyring af krav og specifikationer. Hvis noget ændres, skal det dokumenteres, og det skal gøres tydeligt, hvad der er nyt. Samtidig skal der være en ansvarlig, der sikrer, at ændringer kommunikeres til hele gruppen.

I starten af projektet lavede vi i fællesskab en samarbejdskontrakt, hvor vi aftalte forventninger til arbejdsindsats, mødedeltagelse, kommunikation og programmering. Kontrakten blev udarbejdet sammen ved et fysisk møde, hvor alle fik mulighed for at byde ind. Det gav en god fælles forståelse fra start, og vi har generelt formået at overholde de aftaler, vi lavede.

1. Ugentlige måder m. vejleder
2. Typst til skrivning af rapport
3. Sociale gruppe aktiviteter, med jævne mellemrum
4. Discord som primær kommunikationskanal
5. Mødetid: 09:00 -> end of day
6. Gruppe kodning
7. Møder er fysiske (med mindre andet anvist)
8. Meld sygdom, forsinkelse, mm. tidligst muligt
9. I starten primært møder når anden undervisning henligger, tilrettes efter behov
10. 1st fortagelse forbigås, 2nd fortagelse er gruppesnak, 3rd foretagelse er vejleder.
11. Strengt profesionelle måder (pauser tilladt)
12. Discord kanalen skal holdes ren og profesionel
13. Referat af møder hvis gruppemedlem ikke tilstede
14. IDE:
    Frontend: Webstorm
    Backend: free choice

Figure 1: Samarbejdskontrakt

Gruppemøder

For at holde overblik over gruppearbejdet blev der også skabt en skabelon for gruppemøderne. Skabelonen indeholder dagens målsætninger, opgaver for dagen samt lektier m. m. Skabelonen har givet gruppen mulighed for at samle alle opgaver et sted, og er blevet refereret til, hvis man nu glemte hvilke opgaver man skulle lave.

Opstart og opsummering

February 20, 2025

Dagens emne	
Dagens målsætninger	

Opgaver og lektier

Gruppemedlem	Opgave i dag	Lektie færdig?	Lektie til næste gang
Sebastian			
Malte			
Lucas			
Jumana			
Ibtisam			
Stinnia			
Fælles			

Afrunding og opsummering

Afrunding	
Næste møde	

Figure 2: Gruppemøde skabelon

Samarbejde med vejleder

Samarbejdet med vores vejleder i P2 har generelt fungeret godt. Vi har haft regelmæssige vejledninger, når der har været behov for det, og vejlederen har været tilgængelig og støttende i forhold til vores spørgsmål og usikkerheder. Det har skabt tryghed og hjulpet os med at få retning i projektet, især når vi stødte på udfordringer. Dog har vi erfaret, at der i visse faser af projektet kunne have været gavn af mere løbende dialog. Til P3 overvejer vi derfor at planlægge hyppigere vejledninger, så vi sikrer en mere jævn udveksling af input og kan reagere hurtigere, hvis noget begynder at skride i den forkerte retning. Vi har også gjort brug af et vejlederskema, hvor vi har skrevet vores spørgsmål ned inden vejledning og efterfølgende noteret den feedback, vi har modtaget. Det har fungeret rigtig godt som en fælles opsamling, især hvis nogen i gruppen har været fraværende under vejledningen. På den måde kunne alle efterfølgende læse, hvad der blev drøftet, og hvad vi skulle arbejde videre med. Det har bidraget til en bedre fælles forståelse og kontinuitet i vores arbejde.

Vejledermøder

DD. Month Year

	Kort beskrivelse	Feedback
Sendt til vejleder		

Emne og spørgsmål til vejleder

	Vores spørgsmål	Feedback fra vejleder
Indhold/fokusområder		
Angående rapport		
Angående programmering		
Andre spørgsmål		
Til næste gang		

Figure 3: Vejledermøde skabelon

Socialt samarbejde og trivsel

En af de positive erfaringer fra P2 var, at vi havde det godt socialt. Det skabte en god stemning og hjalp med at løse konflikter hurtigt. Dog kunne vi godt have brugt flere planlagte sociale aktiviteter, så det ikke bare blev noget, der opstod tilfældigt. I P3 vil vi afsætte tid til sociale aktivteter. Det kan fx være en frokost ude efter et møde. Vi vil også være mere opmærksomme på, hvordan vi hver især trives i projektet, og om nogen er ved at køre trætte. Det skal være tilladt at sige fra og bede om hjælp, hvis man er i tvivl om sin opgave. Vores hovedmål for P3 er at arbejde mere struktureret og samtidigt skabe bedre faglig og social sammenhæng. Ved at lære af de fejl og succeser, vi oplevede i P2, har vi et godt udgangspunkt for at få et bedre og mere velfungerende projektforløb i den kommende periode.