Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag

Nils-Jarle Haugen
Erlend Solbakken Nikolaisen
Trygve Johansen Woldseth

Màoyi

Handelsplattform for sjømat

Bacheloroppgave i Dataingeniør
Veileder: Mikael Tollefsen

Mai 2021

NTNU
Norwegian University of
Science and Technology

Nils-Jarle Haugen
Erlend Solbakken Nikolaisen
Trygve Johansen Woldseth

Màoyi

Handelsplattform for sjømat

Bacheloroppgave i Dataingeniør
Veileder: Mikael Tollefsen
Mai 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag

NTNU

Kunnskap for en bedre verden

Forord

Vi ønsker å rette en takk til

- vår oppdragsgiver, Kurt Louis Skjong og James Roger Eide fra Giske Kystfiske AS, for godt samarbeid gjennom hele prosjektet.
- vår veileder, Mikael Tollefsen, for gode råd og tilbakemeldinger gjennom denne perioden.

Sammendrag

Giske kystfikse AS er en av mange fiskebåter i Ålesund som selger sine varer fra kai. De ønsket seg en løsning som vil gjøre det enklere for selgere og kjøpere å komme i kontakt og utføre handel. De ønsket en løsning der fiskerene kunne tilpasse varer etter kjøpernes etterspørsel. Målet med denne oppgaven ble da å utvikle en fullstack-tjeneste for kjøp og salg av sjømat.

Vi valgte å løse dette ved å lage en mobilapplikasjon i Flutter, som kan kjøres på både iOS og Android. Serverapplikasjonen for å drive selve tjenesten implementerte vi ved hjelp av JakartaEE-rammeverket OpenLiberty. Løsningen kjøres som et sett mikrotjenester i Kubernetes. Vi har også implementert et webgrensesnitt i React for å administrere tjenesten.

Resultatet ble en løsning som utgjør en fullverdig handelsplattform som er svært moderne utviklet. Serveren er robust og skalerbar, og tilbakemelding fra oppdragsgiver om mobilapplikasjonen, var svært positiv.

Forkortelser

API Application Programming Interface. 44, 51, 65, 66

DBMS Database management system. 58

ECB Entity Controller Boundary. 13

IPC Inter Process Communication. 52, *Se:* Interprosesskommunikasjon

JSON JavaScript Object Notation. 46, 70

JWT JSON Web Token. 59, 81, 90, 92

POJO Plain Old Java Object. 45

POM Project Object Model. 78

UML Unified Modeling Language. 13, *Se:* Unified Modeling Language

URI Uniform Resource Identifier. 26

Terminologi

autentisering en prosess for å bekrefte en påstått identitet, for eksempel ved innlogging av bruker. 47

best practise anerkjente og dokumenterte aktiviteter eller prosesser som med et vellykket resultat har blitt benyttet av flere organisasjonser (AXELOS ITIL® Publications 2012). 33

callback prosess som returnerer et kall tilbake, slik at den kan brukes andre steder i koden . 71

chown kommando som benyttes i Linux for å sette bruker- og gruppeeierskap. 87

debugging prossessen ved å finner og fikser bugs(feil eller problemer som hindrer korrekt operasjon. 65

exception en unntakssituasjon i Java-programmer, som kan håndteres med en egen type programkode. 47, 92

informasjonssikkerhet sikring av opplysninger ved å bruke prinsippene om konfidensialitet *(sikre mot uvedkommende)*, integritet *(sikre mot utilsiktet eller uautorisert endring/sletting)*, og tilgjengelighet *(sikre at de er tilgjengelige for tiltenkt formål)*, (Datatilsynet 2021). 32

Interprosesskommunikasjon en mekanisme i et operativsystem for kommunikasjon på tvers av prosesser. 52

l10n numeronym for “localization”. **L**, fulgt av 10 bokstaver, og så **N** . 18

open source åpen kildekode som alle kan bruke og endre på. 65

protokoll en spesifikasjon på hvordan kommunikasjon skal foregå. 22

service discovery prosess som automatisk oppdager og tilgjengeliggjør tjenester som kjøres på et nettverk. 53

Unified Modeling Language en standard som brukes for modellering av data i IT-sektoren. 13

INNHOLD

Liste over Figurer

Liste over Kodeeksempel er

Del 1

Innledning

I innledningen vil vi redegjøre for oppdragsgivers beskrivelse av problemstilling i kravspesifikasjonen. Oppgavens oppbygging vil også bli presentert her.

1.1 Bakgrunn og problemstilling

Giske kystfiske AS er en av mange fiskebåter i Ålesund som selger sine varer fra kai. Selv om denne metoden fungerer, er den tidkrevende og uforutsigbart. Fiskerne vet ikke hva kunden ønsker å kjøpe, og vet derfor ikke hva de skal fiske. Det beskrives fra Giske kystfiske AS at de ofte sitter igjen med usolgt fangst, samtidig som de ikke kan tilby kjøperne det de spesifikt ønsker av type fisk.

1.2 Formål

Formålet var å utvikle en full-stack applikasjon for handel av sjømat. Oppdragsgiver ønsket en applikasjon der fiskere kunne legge ut fangsten sin, og kjøpere kunne legge inn bestillinger. Dette vil medføre at flere kunder får kjøpt hva de ønsker, siden fiskeren kan se hva kundene etterspør. Målet er at fiskeren da i større grad vil kunne tilby varer etter etterspørsel. Færre selgere vil da sitte igjen med fangst siden det blir enklere å gjennomføre handel. Måten denne app-løsningen skal finansieres på er at selgere tegner et abonnement, som gir

tilgang til å legge ut varer på plattformen for salg.

1.2.1 Kravspesifikasjon

Oppdragsgiver har laget en kravspesifikasjon som beskrev hans initielle syn på hvordan mobilapplikasjonen skulle se ut. Kravspesifikasjonen hadde disse funksjonelle kravene:

- Mulighet for å legge ut salg og bestilling
- Norsk og Engelsk språk
- Kart for å vise hentested ved salg
- Rating av motpart
- Fungere på iOS og Android

og disse ikke funksjonelle kravene:

- Kunne brukes på telefon og nettbrett
- Høy sikkerhet

Den fullstendige kravspesifikasjonen ligger som vedlegg D.

1.3 Avgrensninger

Generelt i dette prosjektet har vi hatt stor valgfrihet og det har vært lite avgrensninger (med tanke på valg av teknologi). En avgrensing som påvirket oss var at oppdragsgiver spesifiserte at mobilapplikasjonen måtte fungere med både iOS og Android. For å kunne fullføre prosjektet innenfor tidsrammen, valgte vi å bruke et kryssplattformrammeverk.

1.4 Rapportens struktur

Her beskriver vi hvordan resten av rapporten er strukturert.

Del 2 - Teoretisk grunnlag:
Del 2 forklarer teorien bak konseptene og teknologiene vi har brukt i dette prosjektet.

Del 3 - Materialer og Metode:
Del 3 beskriver de materialene og metodene vi har brukt i prosjektet. Dette er ting som teknologier og programvare.

Del 4 - Resultater:
Del 4 går gjennom det vi har resultatene vi har oppnådd med dette prosjektet. Applikasjonen og backend-serveren er beskrevet i sin helhet i denne delen.

Del 5 - Drøfting:
I del 5 drøfter vi valg som vi utført i løpet av prosjektet og om vi kunne gjort ting annerledes.

Del 6 - Konklusjon:
I del 6 avslutter vi oppgaven med en konklusjon som oppsumerer resultatene og de erfaringene som vi har gjort oss gjennom prosjektet.

Del 2

Teoretisk grunnlag

I denne delen vil vi beskrive de forskjellige teoretiske grunnlagene vi har basert mange av våre senere avgjørelser i prosjektet på.

2.1 Eksisterende løsninger

Noe av det første vi gjorde mens vi planla dette prosjektet var å se etter eksisterende løsninger som kunne løse problemstillingen oppdragsgiver presenterte for oss. Den eneste løsningen vi fant som lignet på det ønskede produktet, var tjenesten https://bestillefisk.no/. Denne tjenesten består av en nettside hvor man kan bestille fisk og skalldyr, og få de levert på døren. På bestillefisk.no kan ikke fiskere registrere seg for å selge egen fangst, og tjenesten finnes kun som nettside og ikke applikasjon. Dette er viktige punkt som skiller den fra produktet som oppdragsgiver ønsker seg.

2.2 Konsistens og kobling

Konsistens(cohesion) og kobling (coupling) er to sentrale prinsipper innenfor objekt-orientert programvareutvikling, for å oppnå god design på et dataprogram. Med god design menes høy kodekvalitet og god stil, slik at endringer i programmet kan gjennomføres uten unødvendig arbeid. For å enkelt gjøre dette, tilstreber man etter beste evne å oppnå det som kalles “løs

kobling og høy konsistens" på den utviklede koden (Carlson 1996).

2.2.1 Konsistens

Konsistens (cohesion) definerer graden på hvor godt en programdel eller komponent passer sammen etter logikk og funksjon.

2.2.2 Kobling

Kobling (coupling) definerer i hvor stor grad forskjellige komponenter eller elementer i et program avhenger av hverandre.

2.3 Designmønstre

Designmønstre i programvareutvikling er generelle gjenbrukbare løsninger på problemer som ofte oppstår i en gitt kontekst. De er ikke ferdige design som kan gjøres om til kildekode, men er heller beskrivelser på hvordan man løser et problem.

2.3.1 Observer

Observer-mønstret gjør det mulig for et objekt å registrere seg hos og motta notifikasjoner fra en leverandør. Mønstret definerer en "provider" og null eller flere "observers". "Observers" registrerer seg med en "provider" og når en predefinert tilstand eller hendelse oppstår vil "provider" automatisk gi beskjed til alle "observers" ved å kalle en av metodene til dem (Microsoft 2021a).

2.3.2 Singleton

Singleton-mønstret er et programvare-design-mønster som forhindrer mer enn en instans av en klasse å eksistere. Singleton mønstret brukes for å løse problemer som hvordan kan vi

være sikre på at det bare finnes en instans av en klasse og hvordan kan vi enkelt få tak i den instansen (Wikipedia 2021i).

2.4 Entity Controller Boundary

Entity Controller Boundary er et arkitekturmønster som grupperer de ulike delene av et system i tre typer objekter; *entiteter*, *kontrollere* og *kanter*. ECB-mønsteret tar utgangspunkt i funksjonelle krav, og ut ifra disse muliggjør effektiv utvikling av et system med god, logisk oppdeling. ECB-mønsteret gir robust kode, ved å sette regler hvordan objektene i mønsteret kommuniserer i forhold til hverandre. Under utvikling av applikasjonsserveren organiserte vi kildekoden etter ECB-mønsteret. Dette sørger for en oversiktlig og ryddig struktur. Mønsteret er standardisert som egne figurer i Unified Modeling Language (UML)-diagrammer. På sekvensdiagrammene i Del 4.3.5 har vi derfor benyttet disse figurene for å visualisere hvordan vi tok i bruk ECB-mønsteret. Nedenfor forteller vi kort om de forskjellige objektene i ECB (Eclipse Process Framework 2009).

Entity **Entiteter** menes med et passivt objekt, som har ansvar å representere en form for meningsfull informasjon. Entiteter kan kjenne til andre entiteter, men bare kommunisere med kontrollere. Klasser på applikasjonsserveren som representerer entiteter, har *Java-annoteringen* «@Entity».

Control **Kontrollere** håndterer og bestemmer flyt av informasjon i en gitt situasjon, og er bindeleddet som kommuniserer med både entiteter og kanter. På applikasjonsserveren er klassene som styrer informasjonsflyten organisert i mapper med navn «control».

Boundary **Kanter** ligger i periferien eller på grensen til det som definerer et system. Informasjon som flyter inn eller ut av et system, og dermed passerer systemgrensen må gå gjennom en kant. Kanter kan kommunisere med kontrollere og andre kanter. Alle REST API-endepunktene 2.9.4 som utgjør systemgrensen på applikasjonsserveren, er organisert under «boundary»-mapper.

2.5 Kvalitetssikring

Vi trenger å forsikre oss om at koden vår har høy kvalitet. Høy kvalitet på koden, vil gjøre det enklere å implementere nye funksjoner uten at det skaper problemer med eksisterende kode.

2.5.1 Null-sikkerhet

Null-sikkerhet eller *void-sikkerhet* er en garanti i et objekt-orientert programmeringsspråk, om at ingen referanser til objekt vil ha null eller "void" verdier (Wikipedia 2021k). Java har ikke null-sikkerhet som en del av språket men i Java 8 ble det introdusert klassen *Optional* som kan brukes for å sikre seg mot null referanser.

2.5.2 Kodetesting

Kodetesting er en viktig del av programvareutvikling. Det brukes for å forsikre seg om at koden fungerer, både ved forventede og uforventede situasjoner.

2.5.2.1 Unit-testing

Det første nivået av testing er unit-testing. Unit-testing er en type programvaretesting, der individuelle enheter av kildekoden testes for å finne ut om de er klar for å brukes. Unit-tester er ofte automatiserte tester som blir skrevet og kjørt av utviklere, for å forsikre seg om at en del av applikasjonen oppfører seg som forventet (Wikipedia 2021j). Målet med Unit-testing er å isolere hver del av et program, og vise at de individuelle delene kjører som forventet.

2.6 brukergrensesnitt

For å utvikle brukergrensesnitt for mobilapplikasjon og webgrensesnitt på en best mulig måte, har vi støttet oss på den del litteratur og prinsipper som omhandler dette. For at mobilapplikasjonen skal kunne utvikles med flere språk på en kodebase, har vi også tatt i bruk

konseptet lokalisering. I de neste avsnittene forteller vi mer om hva som utgjør gode brukergrensesnitt og hva lokalisering er.

2.6.1 Don Normans grunnprinsipper

Don Normans grunnprinsipper er velkjent innenfor forskning om interaksjonsdesign, og mye referert til i litteratur(Sharp 2019) som benyttes som pensum for flere emner innenfor fagfeltet(UiO 2021), (NTNU 2021). I hovedsak tar prinsippene for seg hva som er viktig når en skal lage gode brukergrensesnitt, som enkelt kan forstås og raskt tas i bruk.

2.6.1.1 Synlighet

Synlighet - *visibility*, beskriver hvorvidt et element i et brukergrensesnitt kan oppfattes av brukeren. For at et element i det hele tatt skal tas i bruk, må det ha en synlig plassering. Desto mere synlig et element er, jo raskere kan brukerne oppfatte det. Motsatt, til mindre synlig et element er, desto vanskeligere vil det være for brukerne å ta det i bruk. Derfor er det viktig at elementer med stor betydning og avgjørende funksjonalitet, har en sentral, synlig plassering i brukergrensesnittet. Godt eksempel på plassering kan være viktige elementer i en bil, som nød blink-bryteren. Underveis i utviklingsarbeidet har vi fokusert på at viktige elementer lett skal kunne oppdages, som oversikt over varekategorier i mobilapplikasjonen.

2.6.1.2 Tilbakemelding

Tilbakemelding eller *feedback* omhandler å sende tilbake informasjon til brukeren om at en handling har blitt utført. Dette kan være visuelt i form av bevegelse eller lys, taktilt *(noe som er følbart)*, lyd eller en kombinasjon av alle disse. Det er kritisk at tilbakemeldingen gis umiddelbart (Norman 2013, s. 23). Ved for lang forsinkelse, gir brukeren opp og finner noe annet å gjøre. I verste fall kan det lede til sløsing av ressurser, siden brukeren har forlatt systemet når handlingen er utført. Mengden tilbakemelding må også være velbalansert; for mye kan føre til irritasjon og ignoranse. Tilbakemelding er tett knyttet opp mot synlighet.

2.6.1.3 Relasjoner

Relasjoner eller *mapping* er lånt fra matematikken og beskriver forholdet av plasseringen mellom forskjellige elementer. For at et brukergrensesnitt skal være enkelt å ta i bruk, er det viktig å basere seg på naturlige relasjoner. Med naturlig relasjon menes det at funksjonen et element har, er logisk representert etter plasseringen(Norman 2013, s. 21). For å danne oversikt, plasseres ofte elementer i grupper som har relevante funksjoner. Et godt eksempel på dette kan være pil-tastene på et tastatur; pil opp er plassert over pil ned, og pil venstre/høyre er plassert naturlig. Ved trykk på pilene, vil dette i et skriveprogram flytte markøren etter retningen på pil-tastene. Det er også sentralt at elementene gjør det brukeren forventer. Ved at naturlige relasjoner er selvforklarende, vil brukerne enklere kjenne igjen grensesnittet.

2.6.1.4 Konsistens

Med konsistens eller *consistency* menes det at elementer i et brukergrensesnitt som ser like ut og som har tilnærmet samme framgangsmåte, vil føre til at en beslektet handling utføres. Et konsistent grensesnitt baserer seg på faste regler på hvordan handlinger utføres. Et slikt grensesnitt er enkelt å ta i bruk, fordi nye brukerne trenger bare å kjenne til den spesifikke regelen som grensesnittet følger (Sharp 2019, s. 29). Godt eksempel på dette er hvordan klikk-funksjonen for musepekeren fungerer likt på tvers av dataprogrammer; et venstre-klikk gjør valg, og høyre-klikk viser liste over mulige valg.

Konsistens er tett knyttet opp mot synlighet og begrensinger; det fungerer godt i grensesnitt med få elementer. Derimot ved komplekse grensesnitt med mange elementer, vil det være utfordrende for brukerne å finne frem til hvilket element som gjør hva. I slike situasjoner vil det være bedre å gjemme bort elementer i undermenyer, selv om dette kan føre til et grensesnitt med lavere konsistens. For å enkelt finne gjemte elementer, er det kritisk at fremgangsmåten for å gjøre nettopp det - er konsistent og følger samme regel.

2.6.1.5 Muligheter og signaler

Gode brukergrensesnitt er selvforklarende; brukeren kan ta de i bruk uten noen særlig form for opplærling. Dette muliggjøres i form av at elementene i grensesnittet gir hint til hvordan de kan benyttes og hvor handlingen finner sted. Dette betegnes på fagspråket som muligheter og signaler.

Muligheter eller *affordance* er relasjonen mellom et fysisk objekt og en person. Eller mer presist en *agent*, noe/noen som kan agere/gjøre noe med objektet (Norman 2013, s. 11). Graden av mulighet bestemmes av både agenten og objektet. For eksempel en tom lenestol gir hint om at den gir støtte, og derfor kan sittes i. De fleste stoler kan også bæres av en person, men svært tunge stoler kan bare løftes av personer med tilstrekkelig styrke. I de tilfeller der det er vanskelig å vite hvor en handling utføres, kan dette kommuniseres til brukeren i form av et signal.

Signaler eller *signifiers* viser hvor en handling kan gjøres. Dette kan være i form av et visuelt merke, lyd eller hvilken som helst indikator som kan oppdages (Norman 2013, s. 14). Et godt eksempel på viktigheten av signaler, er en dør uten handtak. Mangel på handtak tydeliggjør muligheten for at døren kan dyttes, men ikke hvor dette gjøres. For å vise hvor døren dyttes, kan dette løses med et signal i form av et skilt med teksten «SKYV».

Annet eksempel kan være en smarttelefon. Skjermflaten og mangel på fysiske knapper gir hint om mulighet for trykking. Men for å synliggjøre for brukerne hvor de kan trykke, er det behov for visuelle signaler som viser dette.

2.6.1.6 Begrensinger

Begrensinger eller *constraints* omhandler å redusere antall valg en bruker kan utføre på et gitt tidspunkt. Et brukergrensesnitt med få og tydelige valg, gjør at brukeren raskt kan tolke hva som er mulig. Videre fører dette til at beslutninger enkelt kan fattes. På motsatt vis vil mangel på begrensinger og for mange valg, lede til at brukeren begår feil valg. Brukere klander ofte seg selv i prosessen, og korrekte beslutninger fattes så basert på erfaringer av de feil

man gjorde (Norman 2013, s. 65, 125). Begrensninger er knyttet tett opp mot relasjoner og konsistens, som nevnt tidligere. For god brukeropplevelse, har vi underveis i utviklingen lagt vekt på at hver av side skal ha så få valg som mulig.

2.6.2 Lokalisering

Også uttrykt som l10n . Lokalisering er hele prosessen av å tilpasse et produkt til en spesifikk lokasjon. Oversettelse er en av delene som inngår i lokalisering. Utenom oversettelse kan lokaliserings-prosessen også inneholde:

- Å endre design og oppsett for å korrekt vise den oversatte teksten.
- Endre på formatet for dato og tid.
- Endre innholdet slik at det passer bedre til innbyggerne på en lokasjon.

(Globalization og Association 2021)

2.7 Arbeidsmetodikk

Før vi begynte med utviklingen på prosjektet så vi på ulike arbeidsmetodikker for programvareutvikling, som fossefallsmetoder og “agile” utvikling. Vi fant ut at “agile” utvikling var den beste metoden for oss, fordi den er tilpasset for at endringer i planen kan skje underveis i utviklingen.

2.7.1 «Agile» utvikling

“Agile” utvikling er et uttrykk som brukes for å beskrive iterativ programvareutvikling. Det er et uttrykk som omfavner ett sett med rammeverk og praksiser, som er basert på verdiene og prinsippene beskrevet i *Manifestet for Smidig Programvareutvikling* (Agile Alliance 2015). Manifestet ble utformet av sytten personer og er basert på deres erfaring fra å utvikle programvare. Dette er punktene i manifestet:

- **Personer og samspill** fremfor prosesser og verktøy
- **Programvare som virker** fremfor omfattende dokumentasjon
- **Samarbeid med kunden** fremfor kontraktsforhandlinger
- **Å reagere på endringer** fremfor å følge en plan

Punktene til venstre har høyere verdi enn punktene til høyre (Beck mfl. 2015).

2.8 Programvarearkitektur

Programvarearkitektur er måten de forskjellige delene av et program er satt sammen. Boka *Software Architecture in Practice* definerer programvarearkitektur på denne måten:

> *Programvarearkitekturen til et program eller databehandling er systemets struktur eller strukturer, som består av programvareelementer, de ytre synlige egenskapene til disse elementene, og forholdet mellom dem (Bass, Clements og Kazman 2003, s. 21).*

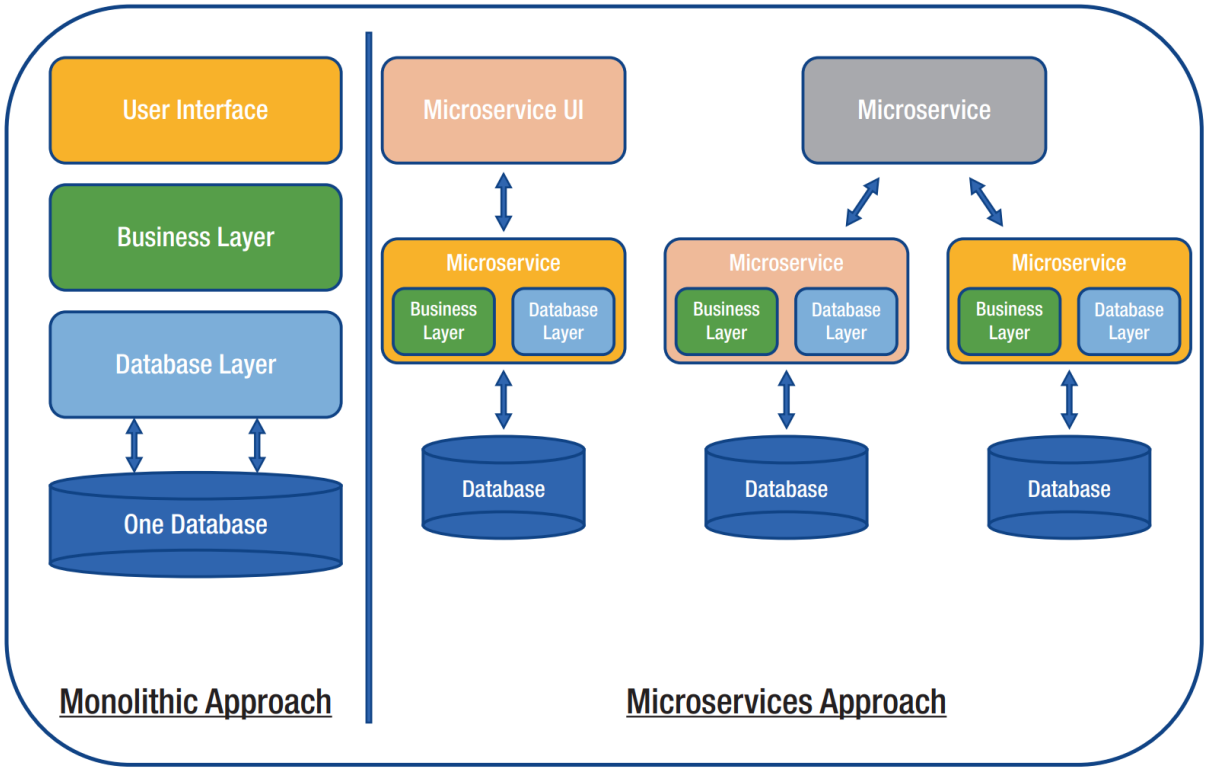
2.8.1 Monolitt

I en monolittisk arkitektur er hele logikken til en applikasjon inni en samling, eller flere samlinger distribuert som en enhet. Med en monolittisk arkitektur, er modulene til en applikasjon tett koblet og gjensidig avhengige (Harsh og Hemant 2019, s. 4). Monolittiske arkitekturer blir ofte delt opp i tre lag:

- Presentasjonslaget: Brukergrensesnitt som lar brukeren interagere med applikasjonen.
- Business-laget: Ryggraden til en monolittisk applikasjon. Inneholder businesslogikken, validering, og manipulering av data.
- Data-laget: Inneholder logikken for henting og "persisting" av data til en *back-end* datakilde fra komponenter lenger opp, som business-laget.

2.8.2 Mikrotjenester

Mikrotjenester er en arkitektur hvor en applikasjon blir delt opp i mindre sett av løst koblede tjenester. Mikrotjenester implementerer en liten mengde funksjonalitet som gjør at kodebasen blir liten og minsker omfanget av en *bug* (Dragoni mfl. 2017, s. 198). I de neste avsnittene går vi gjennom noen av de viktigste konseptene som utgjør en mikrotjenestearkitektur. Figur 2.1 viser forskjellen i oppbyggingen av en monolittisk applikasjon og mikrotjenester.



Source: (Harsh og Hemant 2019, s. 11)

Figur 2.1: Monolittisk arkitektur vs. mikrotjenesete arkitektur

Tradisjonelle monolitter fungerer tilfredsstillende ved mindre applikasjoner, men ved økende grad av funksjonalitet blir de vanskeligere å håndtere. Ny funksjonalitet i komplekse systemer krever stor grad av testing, for å kvalitetssikre at endringer ikke påvirker andre deler av systemet. Mikrotjenester er attraktive fordi de gir stor separasjon, som frikobler funksjonalitet og ansvar mellom tjenestene (Fowler og Lewis 2015).

Ved at hver tjeneste er separert, fører dette naturlig til mindre kompleksitet, som kan håndteres av selvstendige, tverrfaglige team. Tverrfaglige team kombinerer de tradisjonelle rollene

utvikling, testing, kvalitetssikring og drift. Fordi hver tjeneste blir mindre krevende å utvikle mot, og kommunikasjon mellom de forskjellige rollene kan gjøres på en enkel måte, er det mulig at kode raskt kan settes i produksjon. Dette fører til smidighet, og er svært verdifullt ved behov for ny funksjonalitet eller feilrettinger (Amazon Web Services 2021).

En annen fordel med denne separasjonen, er at hver sin mikrotjeneste har sitt eget kjøremiljø. Dette gjør det mulig at forskjellige deler av en mikrotjeneste kan implementeres med forskjellige programmeringsspråk og rammeverk, alt etter hva som passer best for den tjenesten. Dette er en fleksibilitet som ikke er mulig å gjøre på en monolittisk arkitektur .

På den andre siden introduserer mikrotjenester kompleksitet knyttet til hensyn som må tas under utvikling av hver tjeneste, og aspekter rundt hvordan kommunikasjon foregår. For eksempel feil som oppstår i en tjeneste, må kunne håndteres av de andre som avhenger av den. Det må også planlegges for at flere instanser av forskjellige tjenester prøver å gjøre samme operasjoner samtidig.

2.8.3 Virtualisering

Virtualisering er teknologier som etablerer et ekstra lag med programvare mellom maskinvare og applikasjoner. Dette laget etterligner fysisk maskinvare. På denne måten blir det mulig å fordele ressurser til applikasjoner etter behov, i stedet for å knytte dem til en bestemt prosessor, minneområde, harddisk, eller fysisk forbindelse (*virtualisering – IT* 2020).

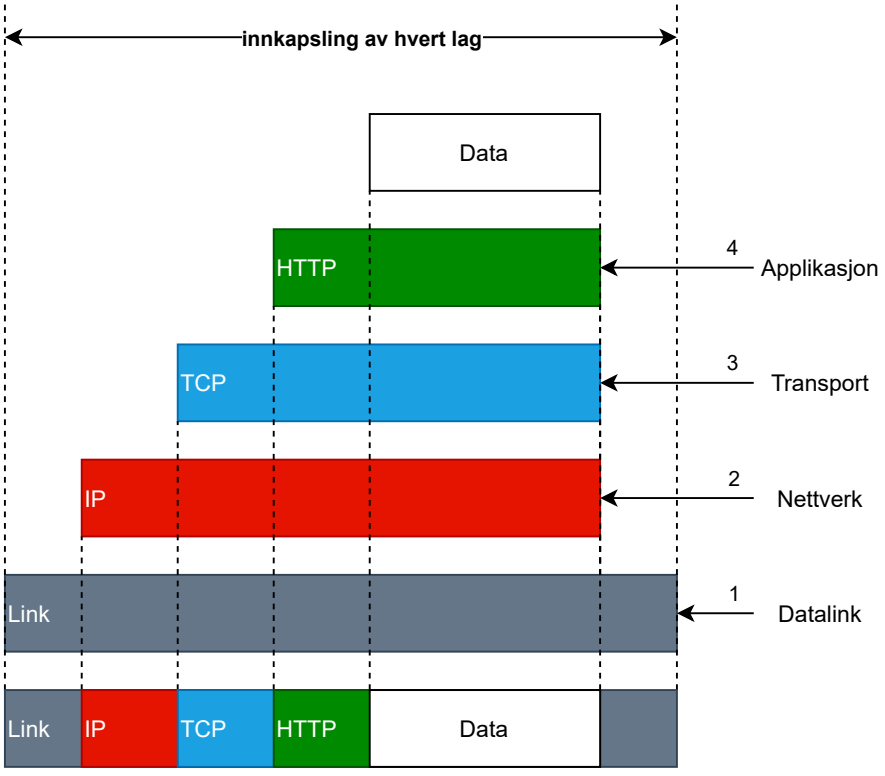
2.8.3.1 Containere

Vi hadde et behov for å kunne kjøre applikasjonen på hver vår datamaskin uten at vi støtte på problemer, fordi vi bruker forskjellige operativsystemer og hadde forskjellig programvare installert. For å løse dette problemet valgte vi å bruke containere. En container er en standard enhet av programvare som pakker sammen kode og alle dets avhengigheter, slik at en applikasjon kjører raskt og pålitelig fra en maskin til en annen (Inc. 2021b). Containere abstraherer applikasjoner vekk fra det miljøet de egentlig kjører på, og lar applikasjonene kjøre på alle systemer uten at spesielle behov må møtes. Containere baserer seg også på en “bruk og kast” tankegang der du kan kjøre opp en container med en applikasjon når du trenger det

og så kan du rive den ned igjen når du er ferdig.

2.9 Nettverkskommunikasjon

For at mobilapplikasjonen skal fungere, er den avhengig av å kunne kommunisere med serveren. Dette muliggjøres ved hjelp av nettverkskommunikasjon. Nettverkskommunikasjon består av å benytte flere nettverksprotokoller, som kommuniserer på forskjellige nivåer av et nettverk. Sammenhengen mellom protokollene kan vises i form av *TCP/IP-modellen*, som definerer de forskjellige nivåene som separate *lag*. De respektive protokollene i nettverket har ansvar for å frakte data frem på sitt *lag*, og de andre protokollene trenger ikke å ta hensyn til hvordan overføringen skjer. Figur 2.2 viser de forskjellige lagene av TCP/IP-modellen.



Figur 2.2: Sammenheng mellom protokoller, her vist ved TCP/IP-modellen

Det vesentlige er at flere protokoller virker sammen for å etablere dette viktige sambandet. Totalt sett utgjør protokollene i felleskap, en logisk forbindelse mellom mobilapplikasjonen og serveren.

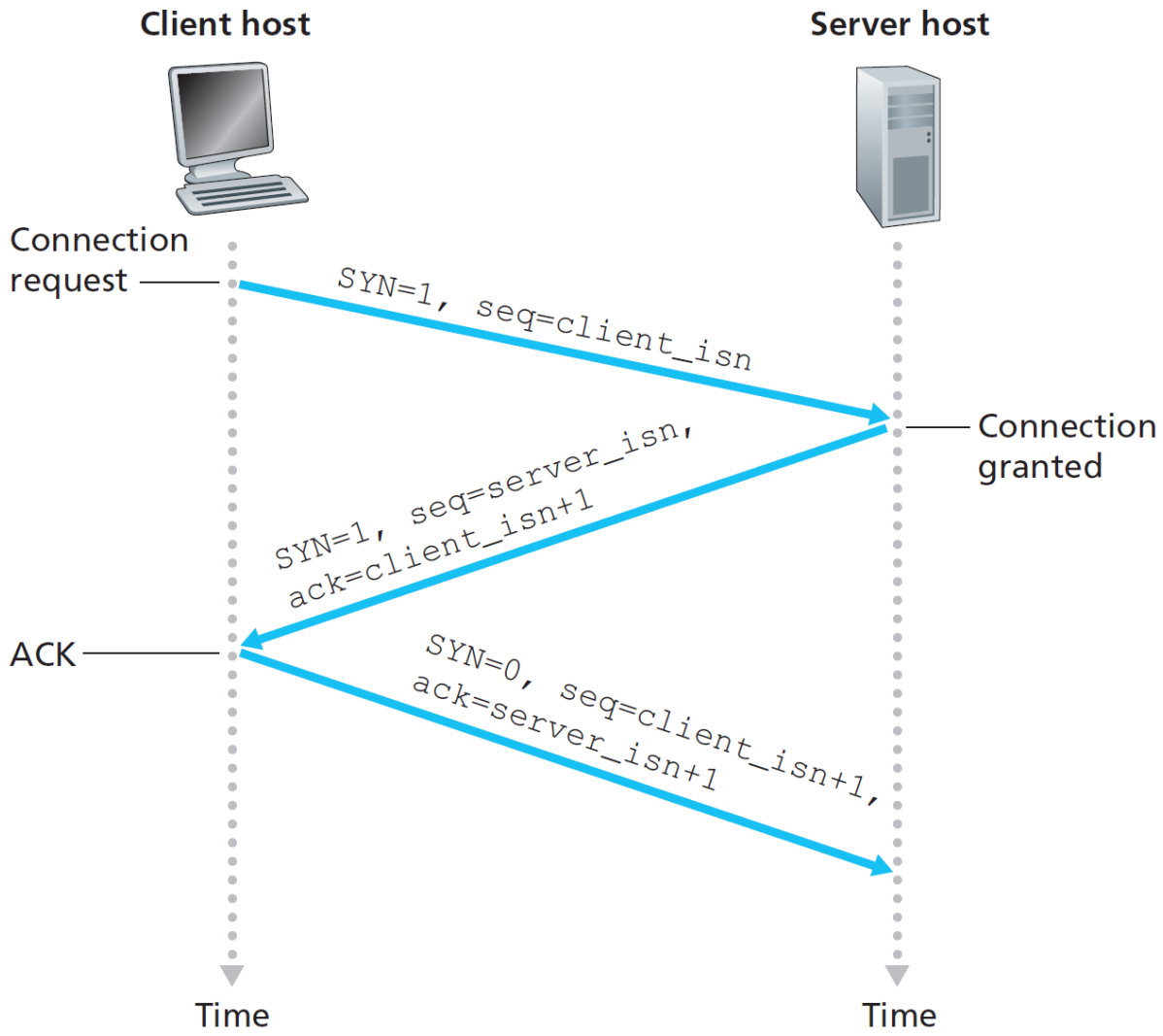
2.9.1 TCP/IP-nettverk

For at vi skal kunne sende data mellom mobilapplikasjonen og serveren benytter vi oss av TCP/IP. Vi bruker TCP/IP for å få pålitelig dataoverføring og adressering av hvor trafikken skal gå. **TCP/IP** er to protokoller som brukes for å sende og motta informasjon over internett, **Transmission Control Protocol (TCP)** og **Internet Protocol (IP)**. Boken *Computer Networking: A Top-Down Approach* kaller de for to av de viktigste protokollene på internett (Kurose og Ross 2017, s. 33).

2.9.1.1 Transmission Control Protocol

TCP er en av de to transportprotokollene som brukes på Internett. TCP tilbyr en forbindelse orientert service til sine applikasjoner. Denne servicen inkluderer garantert levering av applikasjonslag-meldinger til målet og flytkontroll (matchende hastighet for sender/mottaker) (Kurose og Ross 2017, s. 79). Et TCP-segment inneholder flere felt. To av disse feltene er sekvensnummer og bekreftelsesnummer som brukes av senderen og mottakeren i TCP, for å implementere en pålitelig dataoverføringsservice. TCP sin pålitelige dataoverføringsservice sikrer at datastrømmen som en prosess leser ut fra sin TCP-mottaksbuffer er ukorruptert, uten hull, uten duplikater, og i korrekt rekkefølge (Kurose og Ross 2017, s. 272).

For å minske sanssynligheten for falske tilkoblinger benytter TCP seg av det som kalles "three-way handshake". Vi kan se for oss at en klient skal koble seg til en server. Første steg i denne utvekslingen er at klienten lager et TCP-segment med SYN bit-en satt til 1 og med et sekvensnummer (*klient_isn*) som den pakker inn i et IP datagram og sender til serveren. I neste steg mottar serveren datagram-et med TCP-segmentet og sender et tilkobling-innvilget segment tilbake. Tilkobling-innvilget segmentet har et eget sekvensnummer (*server_isn*), SYN bit-en satt til 1, og et ACK felt fylt med *klient_isn +1*. I det siste steget, når klienten mottar dette segmentet, svarer den med ett siste segment som har SYN bit-en satt til 0 og ACK fylt med *server_isn+1*. Når disse tre stegene er fullført, kan klienten og serveren sende segmenter som inneholder data til hverandre (Kurose og Ross 2017, s. 283). Figur 2.3 viser disse stegene.



Source: (Kurose og Ross 2017, s. 284)

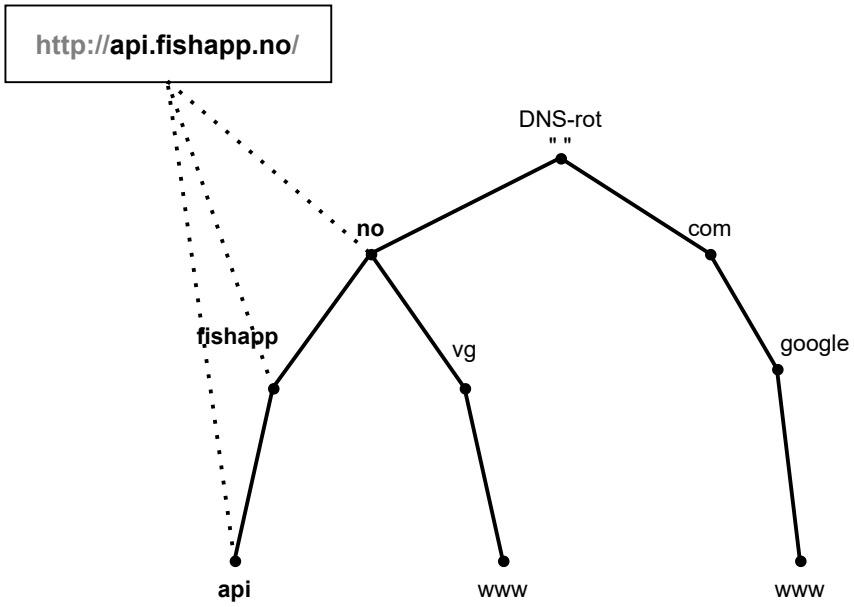
Figur 2.3: TCP three-way handshake

2.9.1.2 Internet Protocol

Internet Protocol (IP) er en protokoll på nettverkslaget, som brukes for å koble maskiner sammen. I dag er det to versjoner av IP som er i bruk, IP versjon 4 (**IPv4**) og IP versjon 6 (**IPv6**). Vi har brukt IPv4. Alle "host"-er og rutere som er koblet til det globale Internett må ha en IP-adresse som er globalt unik (Kurose og Ross 2017, s. 363). Disse IP-adressene brukes av rutere for å finne ut av hvor datagram-ene de mottar skal sendes videre. For å kommunisere med en spesifikk server, angir man IP-adressen til denne serveren.

2.9.2 Domenenavnsystemet DNS

DNS er en tjeneste som oversetter adresser i TCP/IP-nettverk til domenenavn. DNS er en forkortelse for *Domain Name System* og er en svært viktig tjeneste for at Internett og webapplikasjoner skal fungere. Tjenesten er oppbygget i form av et hierarki av mange DNS-servere. 13 rot-DNS-servere rundt om i verden ligger på toppen av hierarkiet, og danner fundamentet for tjenesten. For at en server eller webapplikasjon skal enkelt kunne brukes av andre på Internett, er det nødvendig å ha et domenenavn som en kan knytte til tjenesten. Ved implementasjon av serveren for handelsplattformen, har oppdragsgiver kjøpt et domene som tjenesten kan driftes i fra.



Figur 2.4: Oversikt over oppbygging og arv i DNS, her eksempel ved *api.fishapp.no*.

DNS kan visualiseres som en trestruktur med noder som representerer *DNS-sone*. Hver sone er delt opp med punktum og tilsammen utgjør alle sonene et domenenavn, vist ved figur 2.4. Eierskap for et domenenavn delegeres og arves nedover fra hver node. Ved eksempel vist i figur vil domenenavnet *fishapp.no* være en del av sonen for *.no*. Den som eier et domenenavn, har kontroll på sin egen *DNS-sone*. Ingen andre enn eieren kan opprette DNS-pekere knyttet til det domenenavnet. Eieren kan også selv lage flere underdomener for sitt eget domene. Bare et bestemt sett med DNS-servere, definert som *autorative*-servere, kan lage DNS-pekere innad i *DNS-sonen* for domenet (Liu 2006, s. 22). Hvis domenet *fishapp.no* skal endre DNS-servere, må dette meldes inn til NORID som drifter de *autorative* DNS-serverne for *.no*-

sonen (NORID 2021).

Ved å eie et domenenavn for applikasjonsserveren, har en ikke behov for eie en spesifikk IP-adresse. Skulle IP-adressen til serveren endre seg, kan domenenavnet til serveren raskt endres ved å oppdatere DNS-pekeren på domenet. Eksempel på når dette kan være gunstig er ved bytte av serverleverandør, eller omdirigering av trafikk ved testing av serveren. Ved endring av en DNS-peker, vil dette etter hvert bli spredt til alle DNS-serverne på Internett.

Domenenavn og DNS forenkler utvikling og drift av webtjenester betraktelig. En slipper å forholde seg til IP-adresser i koden for applikasjonen som lages, og kan istedenfor benytte domenenavn. For å være sikker på at det faktisk er applikasjonsserveren en kommuniserer med, kan det settes opp kryptering med TLS 2.11.1.1, som knytter identiteten til domenenavnet sammen med et sertifikat utstedt til serveren.

2.9.3 Hypertext Transfer Protocol

HTTP eller Hypertext Transfer Protocol er en nettverksprotokoll som benyttes for å etablere en kommunikasjonskanal mellom servere og klienter i et TCP/IP-nettverk 2.9.1. Som nevnt i 2.9.4 benytter vi et REST API-grensesnitt for at server og mobilapplikasjon skal kunne kommunisere med hverandre. Dette tar i bruk HTTP som underliggende transportmekanisme. Hver eneste melding sendt over HTTP, er enten en *HTTP-forespørsel* eller en *HTTP-respons* (Fielding og Reschke 2014). Vi forteller mer hvordan disse fungerer i egne underkapitler.

2.9.3.1 HTTP-forespørsler

I HTTP foregår kommunikasjon i form av operasjoner som kan gjøres på *ressurser*. Forskjellige operasjoner utføres med forskjellige HTTP-forespørsler. Derfor er de ulike typene forespørsler definerte som verb, basert på hvilken handling det ønskes utført. Med en ressurs i HTTP menes en definert plassering (*endepunkt*) på serveren som holder på data. Ressurser er identifisert med en Uniform Resource Identifier (URI). Hvilken type data som er lagret på en ressurs, er opp til implementasjonen. Nedenfor forteller vi mer om hvordan de vanligste operasjonene fungerer.

GET Hente data

Denne forespørselen benyttes av klienter, og overfører serveren sin nåværende representasjon av den etterspurte ressursen. Når det snakkes om å motta HTTP-kommunikasjon, er det GET-forespørsler det menes. For mobilapplikasjonen har vi benyttet GET der det trengs å hente data, som bilder etc., ned fra server.

POST Sende data

En slik forespørsel benyttes når en klient ønsker å overføre data til en server. Spesifikt instruerer POST mottakeren at om å prosessere data i forespørselen som ble sendt. POST er vanlig å benytte når en trenger å sende ny informasjon; som opprettelse av en bruker eller en salgsannonse.

PUT Overskrive data

PUT-forespørsler benyttes for å overskrive data for en ressurs som er på serveren. Til forskjell fra POST, så må ressursen eksistere på serveren fra før. All data eksisterende data om ressursen blir slettet, så data som ikke skal endres må også være med i forespørselen.

PATCH Overskrive deler av data

I motsetning til PUT, er definisjonen for PATCH å overskrive *delstykker* av eksisterende data (Dusseault og Snell 2010). Dette er en stor fordel, fordi en slipper å sende eksisterende data på nytt. I mobilapplikasjonen har vi benyttet PATCH for endring av passord.

DELETE Slette data

En DELETE-forespørsel brukes for sletting av data som ligger på den etterspurte ressursen. En server bør svare med spesielle responskoder i denne sammenhengen. *202 Accepted* hvis handlingen snart vil påbegynnes. *204 No Content* hvis handlingen er startet og klinten ikke trenger mer informasjon. Eller *200 OK* hvis slettingen er startet, og responsen inneholder informasjon om status.

2.9.3.2 HTTP-responskoder

I kommunikasjon som benytter HTTP, vil hver eneste respons eller svar fra server sendes med en kode. Kodene som benyttes er standardiserte i *RFC7231*, og forteller hvilken type respons det sendes tilbake (Fielding og Reschke 2014). Innholdet i selve responsen er definert

av serveren som bygger responsen. For kommunikasjon mellom mobilapplikasjon og server, har vi i utstrakt bruk benyttet statuskoder. Dette hjelper oss med hva vi skal gjøre i ulike situasjoner, som for eksempel fange opp og håndtere feil. Under er de kodene vi har benyttet i API-grensesnittet for serveren for handelsplattformen:

200 OK

Benyttes for å fortelle klienten at forespørselen ble behandlet som den skulle på serveren, og klienten kan forvente å få korrekte data tilbake.

401 Unauthorized

Benyttes for å fortelle klienten at serveren nektet å gjennomføre forespørselen på grunn av manglende *autentisering*. Med dette menes det at klienten som sendte forespørselen ikke kan bevise identiteten sin.

For eksempel hvis en bruker som ikke er innlogget, prøver å hente informasjon fra en side i mobilapplikasjonen som krever at man er innlogget. I slike tilfeller vil serveren sende denne koden, og vedkommende blir sendt til påloggingsbildet.

403 Forbidden

Benyttes for å fortelle at serveren nektet å gjennomføre forespørselen, fordi klienten ikke har tilgang til ressursen det ble spurt om.

Hvis en bruker i mobilapplikasjonen prøver for eksempel å sette nytt passord, og skriver gjeldende passord feil - vil serveren sende denne koden, og brukeren vil bli fortalt at passordet må korrigeres

404 Not Found

Sendes av serveren hvis klienten etterspurte en ressurs eller data som ikke finnes.

500 Internal Server Error

Denne benyttes når det skjer en feil på serveren, som forhindrer at forespørselen kan fullføres.

Når denne koden sendes, vil f.eks. mobilapplikasjonen oppfatte at det har skjedd en feil og fortelle dette til brukeren på en fattet måte.

504 Gateway Timeout

Denne er i bruk der en benytter serverer som håndterer forespørsler på vegne av andre servere. Koden sendes til klienten hvis det ikke er mulig å få svar fra serveren forespørselen skulle gjennomføres på. I mikrotjeneste-arkitekturer vil *API Gatewayer* eller *Reverse Proxy*-servere sende denne hvis mål serveren ikke gir svar.

2.9.4 Residual State Transfer API

Residual State Transfer API er en metodikk for å kjøre nettverkskall over HTTP 2.9.3. Vi har benyttet denne metodikken ved utvikling av API-ene på serveren. Et REST API følger arkitekturstilen av samme navn (REST), som ble definert av forskeren Roy Fielding i år 2000 (Red Hat 2021). REST er med andre ord ikke en standard, men et sett med begrensninger som utvikleren må rette seg etter. For at et API-skal defineres som *REST-fult* må det ha følgende karakteristikker (Fielding, Roy Thomas 2000):

1. Klient-server arkitektur

   Det må benyttes en *klient-server* som er oppbygget av servere, klienter og ressurser, og forespørsler overføres via HTTP-protokollen 2.9.3.

2. Tilstandsløs kommunikasjon

   Kommunikasjon mellom klienter og servere må være tilstandsløse - ingen informasjon kan lagres mellom spørringer eller svar

3. Støtte for buffering

   Data som overføres skal kunne ha støtte for å lagres midlertidig i en buffer, slik at en slipper å overføre samme forespørsel flere ganger. Ved å bare hente data når det trengs forhindrer det unødvendig dataoverføring og reduserer last på serveren.

4. Uniformt grensesnitt

   Sørge for at data som overføres følger en fast regel, ved at ressurser er identifiserbare og separate fra hverandre. Enhver ressurs som kan leses av andre, skal kunne nås fra sine egne URI-er og gi selvforklarende meldinger ved bruk.

5. Mulighet for lagdelt system

Server må ha mulighet for et lagdelt system, som håndterer ulike responser til ulike moduler,uten at klienten trenger å ta hensyn til hvilke moduler som håndterer dette.

(6). *Sende kjørbar kode til klienter*
Mulighet for at server kan sende kjørbar kode til klienten etter behov, som utvider klientens funksjonalitet. Å implementere denne er valgfritt.

2.9.5 Secure Shell

Secure Shell, forkortet SSH er en protokoll som brukes for få tilgang til serverer på en sikker måte. Den baserer seg på en klient/server-modell, og et SSH-serverprogram må være installert på maskinen det skal gis tilgang til. Det skrives ofte bare om SSH som en protokoll, men i praksis finnes det 2 versjoner. SSH-1 som var den opprinnelige protokollen, og SSH-2 som er betydelig sikrere (Barrett 2001, s. 11, 43). SSH-2 ble standardisert i 2006 og er svært utbredt, og er typisk den det siktes til ved omtale av SSH i dagligtale. Felles for begge er at mellom server og klient etableres en kryptert forbindelse. Partene kan garantere hverandres identitet, og kommunikasjon mellom dem ikke kan avlyttes, endres på eller etterlignes av andre. Det kan feilaktig tolkes fra navnet at Secure Shell er et kommandolinjegrensesnitt. Men i virkeligheten etablerer protokollen bare en kommunikasjonskanal, mot et allerede eksisterende grensesnitt/*shell* på serveren. I prosjektet har vi benyttet SSH for å få tilgang til fjerntliggende serverer over nettverket.

2.9.6 SSH File Transfer Protocol

SSH File Transfer Protocol eller SFTP er en kommunikasjonsprotokoll for å overføre filer kryptert. Overføringen gjøres ved hjelp av SSH som underliggende transportmekanisme over nettverket. SFTP har i prosjektet blitt benyttet for kopiering av filer til serverer.

2.9.7 Reverse proxy

For å sende trafikk inn til applikasjonsserveren har vi benyttet en *reverse proxy*-server. Proxyservere har rolle som mellomledd som formidler kommunikasjon mellom to parter. Hver av

partene fører kommunikasjon til proxy-serveren, som videresender informasjonen til den motsatte parten. Ved behov kan også data mellomlagres på proxy-serveren, slik at den ikke trenger å sendes på nytt. Vanlige proxy-servere, kjent som *forward proxy* har først og fremst blitt benyttet til å mellomlagre webinnhold på vanlige klient-nettverk. Når PC-er på et nettverk med proxy-server sender forespørsler til Internett, vil disse i realiteten sendes til proxyen og den vil sende forespørselen videre ut på Internett. Forskjellen mellom *forward proxy* og *reverse proxy* er ikke stor, men svært viktig. En *reverse proxy* er derimot installert i front av et server-miljø. All trafikk fra klienter blir håndtert av proxy-serveren, før den sendes videre til den faktiske serveren som skal motta trafikken. Dette gir stor fleksibilitet for hvordan ekstern trafikk håndteres. Med *reverse proxy* kan trafikk enkelt lastbalansereres, caches, krypteres og ikke minst hindre at den interne strukturen for backenden eksponeres mot Internett (CloudFlare Inc. 2021).

2.9.8 Webhook

Måten betalingsløsningen vi valgte sender tilbake infostatusen til en betaling er gjennom en webhook (*Payment API - DIBS Technical Documentation* 2021). En webhook er en måte å tilby et *callback* til en annen tjeneste ved å ha et endepunkt denne tjenesten kan sende et HTTP kall til.

2.10 Kryssplattformrammeverk

Vi hadde behov for å utvikle en mobilapplikasjone som fungerte både på Android og iOS, og valgte derfor å bruke et kryssplattformrammeverk. Kryssplattformrammeverk er rammeverk som brukes i for å utvikle til flere platformer. I vårt tilfelle benytter vi kryssplattformrammeverk for å lage mobilapplikasjoner som fungerer både på Android og iOS. Dette gjør denne utviklingen enklere. Ved å bruke et kryssplattformrammeverk slipper man å utvikle applikasjonen flere ganger for alle operativsystemene som den skal kjøre på. Man trenger bare skrive koden en gang, så tar rammeverket seg av at den funker på plattformene som rammeverket støtter (*Cross-platform software* 2021).

2.11 Sikkerhet

For å sørge for at informasjonen håndteres av systemet på en trygg måte, har vi gjennom prosjektet tatt i bruk teknikker som baserer seg på teoretisk kunnskap som omhandler sikkerhet. I de neste avsnittene forteller vi om viktige prinsipper vi har tatt hensyn til.

2.11.1 Konfidensialitet

Data som sendes med HTTP 2.9.3 blir overført i *klartekst*. Med andre ord, innholdet kan leses av alle komponenter i TCP/IP-nettverket 2.9.1 trafikken går gjennom - og kan enkelt avlyttes. I et isolert nettverk, der man har kontroll på alle komponenter, kan en klare seg med å benytte vanlig HTTP. Mobilapplikasjonen og serveren kommuniserer i vårt tilfelle over Internett, og vanlig HTTP vil derfor ikke være tilstrekkelig for å hindre at andre avlytter informasjonen.

I teorien om informasjonssikkerhet er konfidensialitet et viktig prinsipp. Datatilsynet definerer begrepet slik:

> «**konfidensialitet.** Prinsipp om at personopplysninger må være sikret mot at uvedkommende får tilgang til dem»

Med *personopplysninger*, menes enhver opplysning som kan knyttes til en enkeltperson (Datatilsynet 2021). Brukere av handelsplattformen oppgir slike opplysninger når de registrer konto fra mobilapplikasjonen. Som et sikkerhetstiltak blir derfor all trafikk mellom mobilapplikasjon og serveren kryptert med TLS 2.11.1.1, slik at den den holdes konfidensiell.

Vi forteller mer om egenskapene til TLS i neste avsnitt.

2.11.1.1 Transport Layer Security

For å forsikre oss om sikker kommunikasjon som går mellom mobilapplikasjonen og serveren valgte vi å bruke *Transpor Layer Security (TLS)*. TLS er en kryptografisk protokoll som er designet for å gi sikker kommunikasjon over et datanettverk. TLS skal gi en sikker kanal som har følgende egenskaper (Rescorla 2018):

- Autentisering: Serversiden av kanalen skal alltid være autentisert. Om klientsiden er autentisert er valgfritt.
- Konfidensialitet: Data som sendes over kanalen skal kun være synlig for endepunktene.
- Integritet: Data som blir sendt over kanalen kan ikke bli modifisert av en angriper uten at det oppdages.

2.11.2 Tokenbasert autentisering

Tokenbasert autentisering er et konsept som går ut på at en bruker kan få et token for å autentisere seg mot en tjeneste. Brukeren logger inn med brukernavn og passord, og får et token i retur. Når brukeren har fått dette tokenet, kan det sendes for å få tilgang til ressurser i en gitt tidsperiode (M. Haekal og Eliyani 2016). I et slikt token står det hvem eieren er og hvilke tilganger de har.

2.11.3 Injeksjonsangrep

Injeksjonsangrep 2.11.3 er en metode for å angripe applikasjonsserveren på Internett. Dette gjøres ved å sende data på en måte som lurer serverprogramvaren til å kjøre ondsinnede kommandoer, som gir tilgang til informasjon uten løyve. Slike *injeksjons-baserte* angrep er svært utbredt, og er per dags dato nummer 1 på OWASP sin topp 10 liste over mest utnyttede sårbarheter for webapplikasjoner (OWASP Top Ten Project 2021). Applikasjoner som benytter relasjonsdatabaser 2.13.1 er utsatt for *SQL-injeksjonsangrep*. Det er derfor sannsynlig at applikasjonsserveren under drift, kan bli rammet med et slikt injeksjonsangrep sendt fra aktører med ondsinnede hensikter. For å forhindre at dette skjer, har vi benyttet JPA-rammeverket EclipseLink 3.7.4.6 til å håndtere forespørsler mot databasen. I utgangspunktet er ikke JPA sikkert uten videre, men EclipseLink garanterer i sin dokumentasjon, at standard-innstillingene for Eclipslink er trygge mot SQL-injeksjonsangrep (Eclipse Foundation 2019a). Vi har derfor benyttet standard-innstillinger og sett på best practise ved implementasjon, slik at applikasjonsserveren i mindre grad er sårbar for SQL-injeksjoner.

2.11.4 Passordsikkerhet

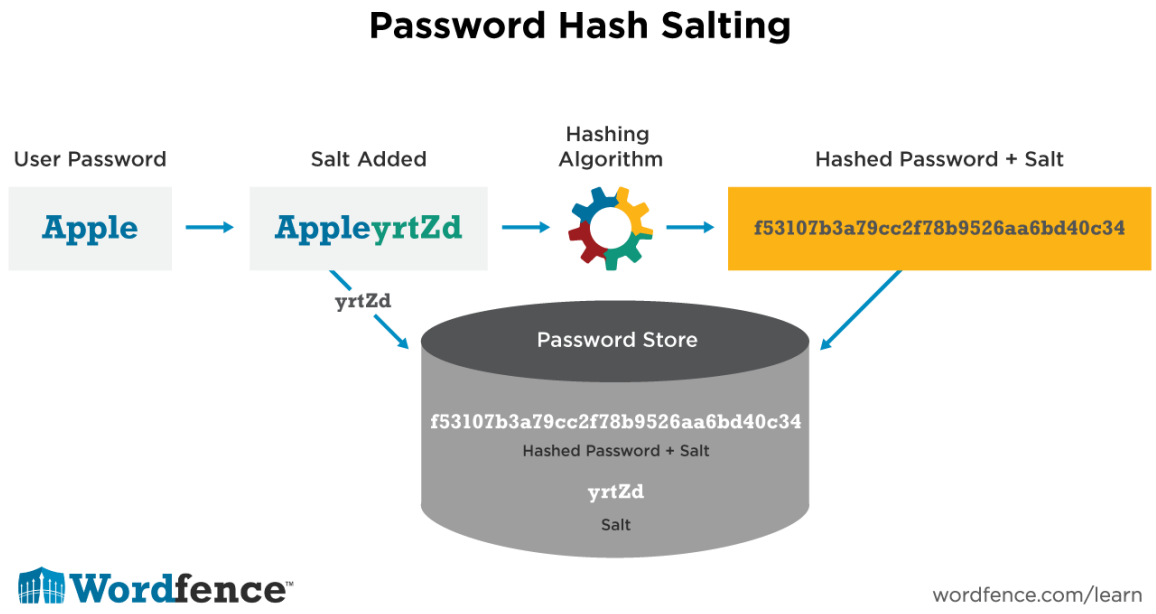
Vi har behov for lagre passord til brukere på en sikker måte. Derfor krypterer vi passordene før de lagres i databasen ved å bruke teknikkene hashing og salting.

2.11.4.1 Hashing

Passord-hashing er en teknikk for å gjennomføre en en-veis tranformasjon av et passord med vilkårlig lengde, til en tilfeldig streng med fast lengde (Chang mfl. 2019, s. 1). En kryptografisk hash-funksjon blir brukt for å omgjøre det gitte passordet, til en streng med tilfeldige tegn med fast lengde som da lagres og brukes som passord. En hash-funksjon blir ofte iterert flere ganger for å forsterke det hashede passordet. Dette vil bremse et angrep med en faktor på 2^{n+m}, hvor n er antall iterasjoner og m er lengden på output i bits (Hatzivasilis 2017, s. 2–3). Hvis mange bruker akkurat det samme passorder vil det hashede passordet også være likt. For å forhindre dette brukes et salt-parameter som hindrer at passord får samme hash-verdi.

2.11.4.2 Salting

Passord salting er en prosess hvor en tilfeldig streng med tegn legges til et passord, før det hashes for å gjøre passord-hashingen mer sikker og vanskeligere å reversere (Kharod, N. Sharma og A. Sharma 2015, s. 3). Ved å legge en tilfeldig salt til et passord vil det samme passordet produsere forskjellige hash-verdier. Vanligvis lagres saltet sammen med det hashede passordet. Ved autentisering kjøres det lagrede saltet og det gitte passordet gjennom hash-funksjonen, og resultatet sjekkes mot den lagrede hash-verdien (Hatzivasilis 2017, s. 3). Figur 2.5 viser hvordan salting av passord gjøres.



Source: (Wordfence 2018)

Figur 2.5: Enkel oversikt over passord kryptering med salt

2.12 Lovverk

Når man utvikler offentlige tjenester eller applikasjoner som kan benyttes av allmennheten, må man ofte forholde seg til en del lovverk innenfor forbrukervern og personvern. Da denne bacheloroppgaven omhandler å utvikle en offentlig mobilapplikasjon, vil dette berøre oss som utviklere for hvordan vi kan implementere systemet. I neste avsnitt forteller vi mer i detalj om hvilke lover vi må ta hensyn til.

2.12.1 GDPR/Personvernforordningen

Siden vi lagrer data om brukere i en database må vi følge *Lov om behandling av personopplysninger (personopplysningsloven)*. Denne loven ble sist oppdatert 15. juni 2018 hvor da *General Data Protection Regulation (GDPR)* ble implementert. GDPR er en forordning som trådde i kraft 24. mai 2018 som regulerer behandling av personopplysninger knyttet til enkeltpersoner i EU.

2.13 Datalagring

Vi hadde behov for å lagre data som ble generert av brukerne av applikasjonen. Vi valgte da å lagre disse dataene i en relasjonsdatabase.

2.13.1 Relasjonsdatabaser

En relasjonsdatabase er en digital database som er basert på relasjonsmodellen. En slik database lagrer ikke bare data, men også informasjon om forholdet mellom disse dataene (Harrington 2009, s. 5).

2.13.1.1 Object-relational mapping (ORM)-rammeverk

Fordi serveren vår benytter seg av objekter for å representere data, trengte vi et verktøy for å oversette objektene til data som kunne lagres i en database. Til dette brukte vi et ORM-rammeverk. ORM er en teknikk for å konvertere data mellom inkompatible type-system ved å bruke objekt-orienterte programmeringsspråk. I objekt-orientert programmering representeres data som objekter og er nesten alltid ikke-skalar verdier. Databaser derimot er ikke objektorientert og kan bare lagre og manipulere skalare verdier, som heltall og strenger organisert i tabeller (*Object–relational mapping* 2021). ORM konverterer objekter til grupper med enklere verdier for å lagres i en database, og motsatt når de hentes.

Del 3

Metode

I denne delen viser vi hvordan prosjektet er satt opp. Dette innebærer alt fra hvordan vi har organisert og planlagt prosjektet til hvordan vi satte opp kjøre og utviklings miljøene. Her vil vi også beskrive hvilke programmeringsspråk og verktøy vi har brukt.

3.1 Involverte partier

Prosjektet besto av tre forskjellige involverte partier. Disse er vist i organisasjonskartet i Figur 3.1. I denne delen beskriver vi hvem de ulike partene er og hvilke oppgaver de har.

3.1.1 Prosjektorganisering

Prosjektorganisasjonen består av tre studenter ved NTNU i Ålesund. Dette er Nils-Jarle Haugen, Erlend Solbakken Nikolaisen og Trygve Johansen Woldseth.

Figur 3.1: Organisasjonskart

3.1.2 Veileder

Faglig veileder representert fra NTNU i Ålesund er Mikael Tollefsen. I løpet av prosjektet skal gruppen få veiledning og råd, slik at gjennomføringen gjøres på en best mulig måte. Vi har holdt jevnlig kommunikasjon med veileder i form av digitale møter annenhver uke og på e-post. Dette har sørget for god oppfølging, og veileder sin erfaring og kunnskap har vært til stor nytte for oss i prosjektet.

3.1.3 Oppdragsgiver

Kurt Louis Skjong er kontaktperson hos Giske Kystfiske AS, og er sammen med James Roger Eide oppdragsgiver for prosjektet. Giske Kystfiske AS utformet den opprinnelige kravspesifikasjonen, som dannet grunnlaget for oppgaven. I gjennomføringen har vi hatt god kontakt med oppdragsgiver i form av kommunikasjon på e-post og fysiske møter. Tilbakemeldingene fra oppdragsgiver har vært nyttige og vi føler samarbeidet har fungert godt.

3.2 Intern organisering

For å effektivt kunne kommunisere og arbeide sammen i prosjektet brukte vi noen forskjellige teknologier. Her beskriver vi hva der er og hvorfor vi brukte de.

3.2.1 Confluence

Confluence er et web-basert wiki/sammarbeidsprogramvare utviklet av Atlassian. Vi bruker det for å samle dokumenter som produktspesifikasjon, “sprint retrospectives”, og møteagendaer og -referater. Vi valgte å bruke Confluence fordi vi har brukt det i tidligere prosjekter, vi har gratis tilgang til det gjennom NTNU, og det kan knyttes sammen med Jira3.3.6.

3.2.2 git

Siden vi er flere utviklere som jobber på delt kildekode var det et behov for å ha et versjonskontrollsystem. Vi valgte å bruke versjonskontrollsystemet *git* (*Git homepage* 2021). Dette fordi det er et mye brukt versjonskontrollsystem som er godt dokumentert. Når man bruker *git*, lagrer en endringene som har blitt gjort i separate enheter som kalles *commits*. Dette gir muligheten til å selektivt rulle tilbake endringer som har blitt gjort, uten å måtte reversere urelaterte deler av prosjektet tilbake til en tidligere versjon. En annen feature av *git* er muligheten for å ha flere *branches*. Dette gir mulighet for å parallelt utvikle flere versjoner av koden og friksjonsfritt bytte mellom disse. Disse branch'ene kan senere trekkes sammen i en *merge*-operasjon hvor endringene blir lagt over i den andre. Det finnes andre versjonskontrollsystemer som Subversion, men siden gruppen allerede var kjent med *git* ble det valgt.

3.2.3 GitHub

For å kunne samarbeide trengte vi et system for å vise og laste ned hverandre sine endringer i *git*. Dette kan løses med et offentlig driftet *git* reposetory hvor alle commit'ene blir lagret. GitHub (Inc 2021a) er en skytjeneste som tilbyr dette. Da har vi muligheten til å arbeide fra forskjellige plasser og kunne enkelt dele koden med andre, som for eksempel veileder. Prosjektgruppens GitHub-repository ligger i Vedlegg I

3.2.4 Pull-request

Pull-request er en funksjon på github (*Github Pull request docs* 2021) hvor under en merge-operasjon, får andre brukere muligheten til å se over, godkjenne eller avise, og kommentere på koden før den blir flettet inn i prosjektet. Pull-requester er hvordan alle nye funksjoner ble lagt til i vårt prosjekt. Dette gjorde at hele grupper ble nødt til å sette seg inn i hva de andre jobbet med, og sikret at det ikke var noe slurvearbeid som kom inn til hoved branchen i prosjektet.

3.2.5 Kommunikasjon

For å sikre godt samarbeid i prosjektgruppen og fremgang i prosjektet, er god kommunikasjon en forutsetning. Grunnet omstendighetene rundt den nåværende pandemien, ble det besluttet å basere seg på at hvert enkelt gruppemedlem arbeider i fra hjemmekontor. En konsekvens av å ikke kunne arbeide samlet, gjør at kommunikasjon mellom gruppemedlemmene blir vanskeligere, som videre kan hemme samarbeidet og i verste fall føre til at fremgangen i prosjektet stagnerer. Det ble derfor tidlig i prosjektfasen klart at gruppen ville trenge et verktøy som kunne håndtere skriftlig og muntlig kommunikasjon, samt videomøter og mulighet for skjermdeling. For skriftlig kommunikasjon mot oppdragsgiver og veileder har vi i stor grad benyttet e-post. For samarbeid innad i gruppen ble det besluttet å bruke tjenesten Discord, mens tjenesten Zoom ble benyttet for skjermdeling og videomøter mot eksterne parter. Ved å ta i bruk to plattformer som Discord og Zoom, vil dette føre til mer robust kommunikasjon, og vi kan ta i bruk den ene som reserveløsning hvis den andre blir utilgjengelig.

3.2.5.1 Discord

Discord er en proprietær kommunikasjonsplattform som støtter tale, tekst, video og skjermdeling. Tjenesten tilbys Discord Inc. (Wikipedia 2021d) både som gratisutgave og en betalingsutgave med noen ekstra funksjoner. Kommunikasjonen skjer isolert i klynger, kjent som“Discord-servere”. Hver registrert bruker på Discord kan være med i flere forskjellige klynger, og på denne måten har Discord blitt et utbredt alternativ for å danne felleskap blant data spillere (Discord Inc 2021). Discord ble tidlig utpekt som passende for prosjektet, av flere grunner. Først og fremst er plattformen brukervennlig, og prosjektmedlemmene har opplevd at det har fungert tilfredsstillende ved tidligere prosjektarbeid. En bestemte seg derfor for å benytte gratisutgaven av Discord for kommunikasjon innad i gruppen, og det ble opprettet en egen“Discord-server”for prosjektet. Skjermdeling og videosamtaler er begrenset funksjonalitet ved den kostnadsfrie utgaven til Discord, og derfor ble det bestemt å bruke NTNU sin Zoom-installasjon til dette.

3.2.5.2 Zoom

Zoom er en plattform utviklet av Zoom Video Communications, Inc (*Zoom Video Communications* 2021) for å holde videokonferanse på tvers av forskjellige samhandlingsplattformer og organisasjoner. NTNU har en egen Zoom-installasjon som studenter og ansatte kan benytte fritt for å opprette videokonferanser (NTNU 2020). Da tjenesten baserer seg på enkelthendelser som møter, i stedet for faste grupperinger som ved Discord, er det enkelt å holde møter med forskjellige parter. Zoom støtter i tillegg til vanlig videokonferanse også skjermdeling svært godt. På dette grunnlaget ble det besluttet å ta i bruk Zoom som kommunikasjonsverktøy internt ved skjermdeling og eksternt for videomøter mot veileder og oppdragsgiver. Om fysiske møter ikke kan holdes, vil Zoom bli benyttet som et alternativ.

3.3 Utviklingsmodellen Scrum

For å organisere arbeidet i dette prosjektet brukte vi den agile arbeidsmetodikken Scrum 2.7.1. I denne delen beskriver vi de forskjellige delene av Scrum.

3.3.1 Sprint

I Scrum er en sprint en arbeidsperiode på 1-4 uker. En ny sprint starter med en gang etter den forrige sprinten er ferdig. Vi valgte å bruke sprinter på 2 uker. I starten av en sprint, tas oppgaver fra en backlog og legges til sprinten. Så fordeles arbeidsoppgavene mellom medlemmene i Scrum-teamet.

3.3.2 Sprint Retrospective

På slutten av hver sprint holder Scrum-teamet et sprint "retrospective" møte. Målet med sprint "retrospective" er å planlegge måter å øke kvalitet og effektivitet. Teamet går over den forrige sprinten og ser på hvordan det gikk med tanke på individer, interaksjoner, prosesser, og verktøy. Scrum-teamet diskuterer hva som gikk bra i løpet av sprinten, hvilke problemer de møtte, og hvordan de problemene ble (eller ble ikke) løst. Scrum-teamet identifiserer de

mest nyttige endringene for å øke effektiviteten (Schwaber og Sutherland 2020c).

3.3.3 Daily Standup

Hver dag i en sprint holdes det "daily standup" møter. Dette er uformelle møter på ca. 15 min. På møte snakker Scrum-teamet om hva de gjorde dagen før, hva de skal gjøre denne dagen, og hvilke problemstillinger de har (Schwaber og Sutherland 2020a).

3.3.4 «User stories»

En «user story» er en uformell, generell forklaring av ønsket funksjonalitet skrevet fra brukerens perspektiv. En "user story" er målet, ikke en "feature", uttrykt fra brukerens perspektiv. Hensikten med en "user story" er å artikulere hvordan et arbeid vil gi en bestemt verdi tilbake til brukeren. "User stories" består maksimalt av et par setninger som sier noe om det ønskede utfallet (Atlassian 2021).

3.3.5 Scrum-team og Roller

Et Scrum-team består av en "Product Owner", en "Scrum Master", og Utviklere. I et Scrum-team har alle medlemmene de ferdighetene som trengs for å kunne skape noe av verdi hver sprint. Medlemmene avgjør internt hvem som gjør hva, når, og hvordan.

3.3.5.1 Scrum Master

"Scrum-Master" er ansvarlig for å etablere Scrum som definert i Scrum-guiden. De gjør dette ved å hjelpe alle forstå Scrum-teori og praksis. Scrum-masteren er ansvarlig for Scrum-teamets produktivitet (Schwaber og Sutherland 2020b).

3.3.5.2 Product Owner

"Product Owner" er ansvarlig for å maksimere verdien til produktet som er et resultat av arbeidet til Scrum-teamet."Product Owner" er også ansvarlig for effektiv bruk av backloggen (Schwaber og Sutherland 2020b).

3.3.5.3 Utviklere

Utviklerne er personene i Scrum-teamet som forpliktet til å produsere en brukbar økning hver sprint. Utviklerne er ansvarlig for å lage en plan for sprinten, gjøre endringer i planen rettet mot sprint målet, og holde hverandre ansvarlig som profesjonelle (Schwaber og Sutherland 2020b).

3.3.6 Jira

For å kunne styret prosjektet valgte vi å bruke Jira for å sette opp Scrum. Jira er et proprietært problemsporings-produkt utviklet av Atlassian, som tillater "bug" sporing og "agile" prosjektstyring (Wikipedia 2021h). Vi valgte å bruke Jira fordi vi har erfaring med å bruke det fra tidligere prosjekter, og vi har gratis tilgang via NTNU. Vi bruker Jira til å sette opp hver sprint, ha kontroll over hvem gjør hva, og hvor langt de er kommet.

3.4 Planlegging av prosjektet

Planleggingen vår for dette prosjektet består av en forprosjektrapport. Som en del av faget *Ingeniørfaglig systemteknikk og systemutvikling* utarbeidet vi en forprosjektrapport som finnes som vedlegg C.

3.5 Kriterer for ferdig prosjekt

Som beskrevet i forprosjektrapporten fikk vi ikke diskutert med oppdragsgiver hva som var kriteriene for ferdig prosjekt før vi startet arbeidet. Vi på gruppen anså prosjektet som ferdig når vi hadde laget system som tilfredsstilte kravene i kravspesifikasjonen. Ut fra kravspesifikasjonen fra oppdragsgiveren laget vi en liste med produktkrav som finnes som vedlegg E.

3.6 Dokumentasjon

For at det skal være lett for andre å forstå og bruke prosjektet vårt, er det viktig at vi lager god dokumentasjon. God dokumentasjon av koden er viktig for at andre lett skal forstå hvordan koden fungerer. Det er også viktig for oss hvis vi kommer tilbake til en del av koden etter å ikke ha jobbet med den på en stund.

3.6.1 Systemdokumentasjon

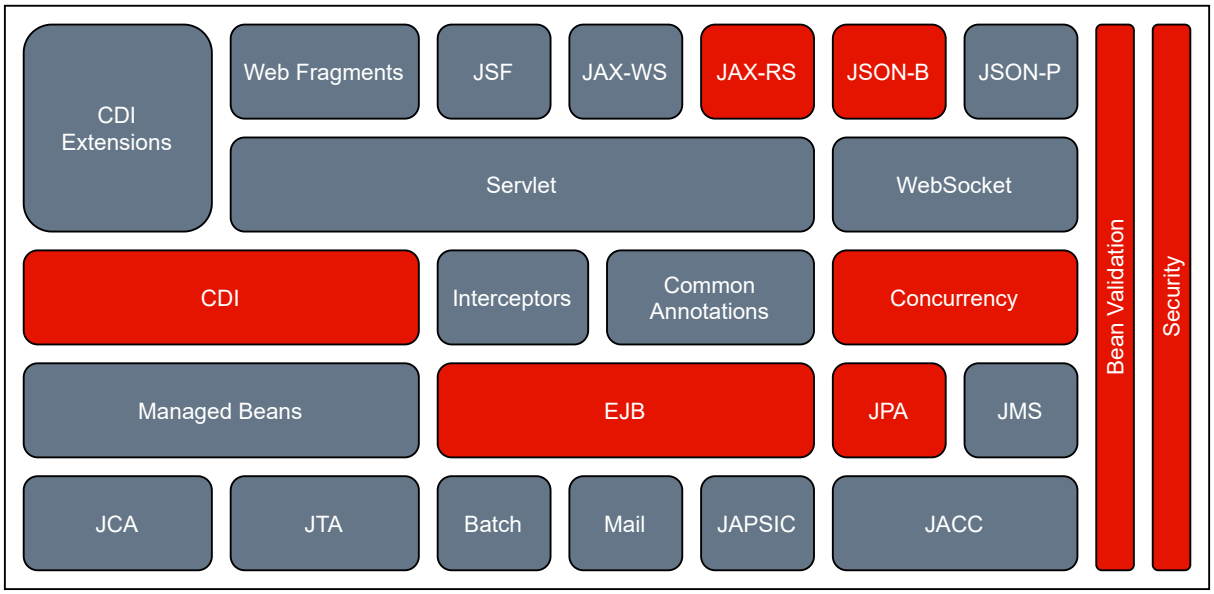
Systemdokumentasjon er dokumentasjon som er assosiert med kildekode, som kan inkludere **README**-filer og API-dokumentasjon (*Software documentation* 2021). Denne dokumentasjonen er mest for andre utviklere, slik at de kan forstå kildekoden og sette opp systemet selv. I vårt prosjekt har vi en **README**-fil som forklarer hvordan man setter opp systemet for at det skal fungere.

3.7 Applikasjonsserver

For å kunne oppfylle kravet om å kommunisere med andre brukere, trengte vi en applikasjonsserver som mobilapplikasjonen kan prate med. Her har vi beskrevet hvordan vi satte opp serveren og hvilke teknologier vi endte opp med å bruke.

3.7.1 Eclipse JakartaEE Platform

JakartaEE er en spesifikasjon vi har benyttet for å utvikle serverapplikasjonen for prosjektet. Rammeverket består av mange komponenter, med mål om å enklere utvikle "enterprise-applikasjoner". Figur 3.2 viser med rødt de delene av JakartaEE som vi har brukt i prosjektet.



Figur 3.2: Komponenter i JakartaEE vi har programmert mot på server (i rødt)

3.7.1.1 EJB

Enterprise Java Beans er komponenter som implementerer business-laget 2.8.1 i en applikasjon. Business-laget er kode som bestemmer selve virkemåten innad i en applikasjon, også kjent som forretningslogikk eller *business logic*. Komponentene skrives som vanlige Java-klasser - Plain Old Java Object (POJO), men har *Java-annoteringer* som sier hvilken type EJB de implementeres som (Goncalves 2009, s. 180). På applikasjonsserveren har vi mikrotjeneste-komponenter som trenger å kjøres hver gang serveren starter. Disse har vi implementert som EJB Session Bean-klasser, med annoteringene *Singleton* og *Startup*.

Det finnes 3 forskjellige typer EJB Session Beans : *Stateless*, *Stateful* og *Singleton*. *Stateless*-typen brukes der det ikke er behov for å lagre eller håndtere tilstand, og enhver instans av den kan brukes når som helst. *Stateful* derimot holder på tilstand, og samme instans må brukes så lenge det er behov for tilstanden. Den siste typen *Singleton*, følger prinsippet 2.3.2 av samme navn - ved at det bare kan eksistere en instans av den i løpet av applikasjonens

kjøretid. Hensikten med dette kan være at EJB-en deles mellom flere klasser, eller at den brukes for initialisering - og skal bare kjøres en gang (Oracle Corporation 2013).

3.7.1.2 JAX-RS

Den delen av JakartaEE spesifikasjonen som beskriver selve REST api 2.9.4-grensesnittet er kalt JAX-RS (Eclipse Foundation 2021e). Spesifikasjonen beskriver et sett med Java annotasjoner som kan brukes for å dirigere HTTP trafikk fra ende-punktet på serveren definert i annotasjonen, til metoden annotasjonen er over.

3.7.1.3 JPA

JPA er en forkortelse for Java Persistence API, og er JakartaEE sin spesifikasjon for Object-Relational-Mapping-rammeverk, som tidligere beskrevet i kapittel 2.13.1.1. JPA benyttes for å håndtere tilstand i en serverapplikasjon. I JakartaEE-baserte applikasjoner beskriver programmeringsgrensesnittet Java Persistance API hvordan tilstand håndteres. Det er flere rammeverk som implementerer JPA, de mest kjente er EclipseLink 3.7.4.6 og Hibernate. I prosjektet ble det bestemt å bruke EclipseLink, fordi den er svært utbredt, men også er referanseimplementasjon for Java Persistence API.

3.7.1.4 CDI

For å opprettholde en ryddig og lettleselig kodebase valgte vi å bruke JakartaEE spesifikasjonen sin avhengighets injeksjon system som heter CDI (*context dependency injection*) (Eclipse Foundation 2021c). Denne spesifikasjonen beskriver et sett med annotasjoner som kan brukes til å beskrive objekter som skal injiseres og andre annotasjoner som beskriver konteksten og derav levetiden disse injiserte klassene vil ha.

3.7.1.5 JSON-B

For å kunne sende Java objektene på serveren over REST kall måtte disse konverteres til JavaScript Object Notation (JSON). Siden JakartaEE-spesifikasjonen definerer en modul som kan

gjøre dette, valgte vi å bruke den. Modulen er kalt JSON-B (Eclipse Foundation 2021d) og er da ansvarlig for hvordan Java objekter blir konvertert til og fra JSON. JSON-B har også støtte for å annotere felt som man eventuelt ønsker å ignorere.

3.7.1.6 Concurrency

Det var situasjoner på serveren hvor vi trengte å spesifisere en oppgave som skulle kjøre på et senere tidspunkt. Det er ikke anbefalt å bruke Java sin innebygde måte, for å kjøre flere tråder på en JakartaEE-server (Juneau 2020).Derfor valgte vi å bruke JakartaEE-rammeverket sin spesifikasjon for parallellisme; "Concurency" (Eclipse Foundation 2021b).

3.7.1.7 Bean Validation

Når data ble mottatt av JAX-RS endepunktene til applikasjonsserveren, trengte vi å validere den. Det vil si å sjekke at den var på korrekt format, slik at informasjon som lagres i systemet ikke er feilaktig, og har tilfredsstillende kvalitet. For å gjøre dette på enkel måte, tok vi i bruk JakartaEE-spesifikasjonen Bean Validation (Eclipse Foundation 2021a). Bean Validation gjør at en kan lage et sett med regler for Java-objekter, ved hjelp av Java-annoteringer. Reglene spesifiserer hvilket dataformat som er lovlig å benytte for den delen av koden som annoteres. Slike annoteringer kan settes på hele klasser eller felt innad i hver klasse. Vi har i stor grad benyttet annoteringene `@NotNull` og `@Valid`. Førstnevnte spesifiserer at et felt ikke må være uspesifisert når et objekt lages. Sistnevnte kjører faktisk validering når et objekt blir initialisert, og går over alle Bean Validation reglene. Følges ikke reglene, stanses prosessen og det kastes en exception.

3.7.1.8 Security

Siden et av kravene for denne oppgaven var høy sikkerhet, måtte vi implementere et sikkert system for autentisering. JakartaEE spesifikasjonen beskriver hvordan et slikt autentiseringssystem med forskjellige roller skal se ut. Denne modulen kalles Security (Eclipse Foundation 2021f). Spesifikasjonen beskriver hvordan autentiseringssystemet med brukere i grupper skal være. Samt et sett med annotasjonen som man kan bruke til å definere hvilke

grupper som kan bruke et endepunkt.

3.7.2 Eclipse MicroProfile

Eclipse MicroProfile er selve fundamentet for at vi i hele hatt har kunnet lage en applikasjonsserver i form av en mikrotjeneste. Prosjektet MicroProfile ble etablert i 2015, og handler om for å gjøre JakartaEE mer portabelt og passende til bruk i mikrotjenester 2.8.2. Det kom i stand som en nødvendighet av at JavaEE, som hadde vært ledende innenfor Java-miljøet de siste 15 år ikke lenger klarte å holde følge med teknologiutviklingen. Oracle hadde trappet ned satsingen på prosjektet (IBM 2019), og utviklingstakten gikk sakte. Det ble klart et behov at JavaEE måtte moderniseres for å holdes relevant.

Dette førte til at utviklere, ledere og leverandører i Java-felleskapet gikk sammen og dannet MicroProfile-prosjektet. De lanserte den første MicroProfile-spesifikasjonen i 2016 (Benevides 2019, s. 37). Siden har flere versjoner kommet til, og MicroProfile 4.0 ble lansert sist i 2019. Målet med MicroProfile-prosjektet er ikke bare å utvikle nye spesifikasjoner, men også sørge for at spesifikasjonene faktisk blir implementert og kan tas i bruk.

Med at MicroProfile skal være portabelt - menes det å ha mulighet for å bytte applikasjonsserver, uten at dette utløser behov for å skrive om koden. Applikasjonsserverer som støtter MicroProfile, har innebygd moduler som implementerer funksjonalitet basert på MicroProfile-spesifikasjoner. På denne måten kan en lage funksjoner som tar i bruk MicroProfile-moduler, uten at disse trengs å lastes ned eller kompileres.

Normalt i Java-prosjekter må man laste ned, kompilere, pakke programmet og alle avhengigheter i et *WAR-arkiv* og så overføre det til applikasjonsserveren. Ved at MicroProfile og JakarataEE -modulene allerede finnes på server - sparer dette først og fremst diskplass, men også tid brukt på kompilering, pakking og overføring av *WAR-arkivet* til server. Noen applikasjonsservere som har implementert MicroProfile er Thorntail fra RedHat, Payara fra Payara Services, TomEE fra Apache og OpenLiberty fra IBM (Eclipse Foundation 2021g).

For applikasjonsserver i dette prosjektet benyttet vi først Payara Server 3.7.4.4. Ved å dermed basere prosjektet på JakartaEE og MicroProfile - muliggjorde at vi midt inni prosjektperioden kunne gjennomføre et bytte over til OpenLiberty 3.7.4.5 som applikasjonsserver. Nedenfor

skriver vi litt om hvilke moduler vi har benyttet fra MicroProfile.

3.7.2.1 MicroProfile Rest Client

MicroProfile Rest Client tilbyr funksjonalitet for å koble til andre webtjenester sine REST API-grensesnitt via HTTP. På denne måten kan man lage komponenter i en mikrotjenestearkitektur 2.8.2 som enkelt kan kommunisere med hverandre.

3.7.2.2 MicroProfile Configuration

MicroProfile Configuration gjør det mulig å skrive innstillinger til applikasjoner på en fleksibel måte. I tradisjonelle applikasjoner settes innstillinger gjerne som statiske variabler i koden før kompilering, eller i form av miljøvariabler på servermaskinen som kjører applikasjonen. Med MicroProfile Configuration kan man definere hvilke variabler man benytter for konfigurasjon ved å markere disse med Java-*annoteringer*. MicroProfile vil videre håndtere hvordan data skrives til variablene. Data kan settes på flere måter, men vil bli lest i prioritert rekkefølge (IBM 2021a), (Eclipse Foundation 2020). Prioritering kan tilpasses, som standard blir data satt med følgende metoder etter prioritet (høyere tall overskriver lavere tall):

400 Java System-properties
Innstillinger som settes i kode av programmereren eller som parametere under oppstart av applikasjonsserveren.

300 Miljøvariabler på systemet
Variabler skrevet til operativsystemet, synlig for alle applikasjoner

100 Filen *microprofile-config.properties*
En fil lagret i filstrukturen til prosjektet. Inneholder innstillinger skrevet i form av variabler. Filen blir pakket som en del av *WAR-arkivet* og er lesbar i Java sin *CLASSPATH*.

— Tilpassede Java-objekter av typen *ConfigSource*
Skrives i koden av programmereren, prioritet blir spesifisert manuelt.

I prosjektet har vi brukt *microprofile-config.properties*-filer og miljøvariabler. Dette er enklere, fordi vi slipper å skrive spesielle start-kommandoer mot applikasjonsserveren, eller kjøre nye kompileringer av programmet når innstillinger skal endres.

3.7.2.3 MicroProfile JWT Auth

For å sjekke identitet til brukerne og gi de tilgang til plattformen, var det behov for å ta i bruk en mekanisme som kunne gjøre autentisering. Det er viktig at implementasjonen som gjør denne viktige sikkerhetsfunksjonen er implementert på en solid måte, og at den aktivt vedlikeholdes. Derfor er det i mange tilfeller vanlig å ta i bruk tredjepartsbiblioteker eller standardiserte implementasjoner som gjør dette. Vi valgte derfor å ta i bruk *MicroProfile JWT Auth* (Eclipse Foundation 2019b) som implementer støtte for token-basert autentisering 2.11.2 med på *JWT* 3.7.4.8.

3.7.3 Kjøremiliø - Kubernetes

For å sette opp og håndtere containerne 2.8.3.1 som utgjør serveren har vi benyttet Kubernetes. Kubernetes er en svært kompleks og omfattende plattform. Derfor forteller vi først om hva Kubernetes er, og deretter hvilke funksjoner vi har tatt oss nytte av i dette prosjektet.

3.7.3.1 Plattformen

Kubernetes er en plattform utviklet av Google som forenkler utvikling og drift av mikrotjenester. Den ble lansert i 2014, basert på et tiår med Googles egne erfaringer for drift av distribuerte applikasjoner (Burns mfl. 2016). Kubernetes består av programvare som danner en plattform for å håndtere store mengder containere på en ukomplisert måte. Plattformen drifter et sett med API-er som styrer et kjøremiljø som containere driftes i. For å sikre høy driftsstabilitet og muligheter for å skalere ressurser, bygges vanligvis Kubernetes som cluster bestående av flere maskiner.

Noder 3.7.3.3 i et Kubernetes-cluster, er enten bygget opp av fysiske eller virtuelle maskiner. Noder kjører de forskjellige containerne i clusteret. Med tanke på oppsett av clusteret

er Kubernetes fleksibel, og man kan melde inn eller ta ut noder forløpende, uten at dette gir nedetid for applikasjonene som kjører i clusteret. Slik kan kapasiteten i Kuberentes enkelt tilpasses til applikasjonene som kjøres i clusteret. Dette gjør at Kubernetes passer godt kjøring av applikasjoner med varierende ytelsesbehov.

Fordi Kubernetes består av standardiserte definerte API-er, kan et lokalt cluster med en maskin, tilby tilsvarende funksjonalitet som et cluster som kjører hos en skyleverandør med flere titalls noder. En utvikler kan for eksempel selv kjøre et Kubernetes-cluster med redusert ytelse, men ha samme muligheter som en skyleverandør. Dette gjør det mulig å kjøre en applikasjon på tilnærmet lik måte lokalt under utvikling, som i produksjon. Mindre forskjeller mellom utvikling og produksjon fører til færre feil. Når feil oppstår, kan disse enklere kan reproduseres og raskere rettes. Velger man å leie Kubernetes i skyen, trenger en ikke å tenke på hvordan clusteret kjøres, og kan fokusere på selve utviklingen av applikasjonen.

3.7.3.2 Konfigurasjon

I Kubernetes spesifiseres alt som utgjør hvordan en applikasjon skal kjøre, før den faktisk startes i gang. Dette kalles *deklarativ konfigurasjon*, og Kubernetes vil alltid sørge for at en applikasjon kjøres på den måte som er spesifisert. Skjer det avvik, som for eksempel at en container krasjer, vil Kubernetes fange opp dette og automatisk starte opp en ny instans (Burns 2019, s. 47).

Infrastruktur og ressurser i en mikrotjeneste defineres ved hjelp av standardiserte API-er. Kubernetes kan styres direkte over HTTP med REST API, med *kubectl*-kommandoer, eller manifester som beskriver tilstand lagret i *YAML*-filer. I prosjektet har vi benyttet oss av et sett med manifest-filer som beskriver tilstanden som utgjør applikasjonsserveren. I seksjon 4.3.4 viser vi hvordan vi har benyttet de forskjellige objektene i Kubernetes.

3.7.3.3 Node

En node er en fysisk eller virtuell maskin som kjører pod'ene 3.7.3.4 i clusteret i.e. selve maskinene som containeren blir kjørt på. Et cluster kan bestå av en eller flere noder.

3.7.3.4 Pod

Pod er en ressurs i Kubernetes som representerer et kjøremiljø for volumer og containere. Pod-er er den minste kjørbare ressursen i Kubernetes, og containere inni en Pod kjøres alltid på samme node. Alle containerne innad i en Pod vil ha samme IP-adresse, som om de var på samme logiske maskin (The Kubernetes Authors 2021). Som et resultat av dette vil alltid Pod-er til enhver tid, kjøres på **spesifikke** noder i et cluster. Med andre ord; Pod-er kan ikke kjøres distribuert på tvers av noder.

Det er vanlig å separere forskjellige applikasjoner ut i Pod-er hvis de kan kommunisere med hverandre over nettverket. I tilfeller der en applikasjon består flere containere som er tett koblet, kan disse kjøres sammen i en Pod. Kommunikasjon i slike tilfeller kan skje via delt minne, delte prosesser som Inter Process Communication (IPC) eller delte volumer. Eksempel kan være en webapplikasjon med to containere, en webserver for å tilgjengeliggjøre filer og en hjelpecontainer som synkroniserer innholdet som deles. På fagspråket kalles slike mindre viktige containere for *sidecar* (Microsoft 2017). En fordel med separate containere, er at de kan prioriteres ulikt hvis det skulle være begrensede ressurser tilgjengelig i clusteret.

3.7.3.5 ReplicaSet

ReplicaSet er en ressurs som brukes for å kjøre et bestemt antall kopier av en Pod på samme tid i et cluster. Det er flere fordeler med dette. For det første gir det **redundans**. Feil som forhindrer en instans av en container i å kjøre, kan tolereres uten nedetid. En annen fordel er at flere instanser kan kjøres på flere noder, og på denne måten kan ressurser til en applikasjon **skaleres**. En tredje fordel er at forskjellige replika av en instans kan kjøre forskjellige oppgaver i parallell, dette kalles *sharding* (Burns 2019, s. 103).

3.7.3.6 Deployment

Vi har benyttet Deployment-ressurser for å spesifisere de forskjellige komponentene på applikasjonsserveren. Et Deployment definerer tilstanden til hvordan applikasjoner skal kjøres i et cluster. I et Deployment blir det spesifisert hvilket Image en container skal kjøre, og hvor mange repliseringer av instansen det skal være. Når et Deployment kjøres i Kubernetes, re-

sulterer dette i at clusteret automatisk oppretter nødvendige ressurser som Pod-er og ReplicaSet. ReplicaSet-et som oppettes vil være konfigurert til å holde det antall Pod-er i drift som er spesifisert i Deployment ressursen. På denne måten slipper man å lage Pod-er manuelt, og Deployment er en vanlig ressurs å bruke for å kjøre applikasjoner.

Konfigurasjon tilknyttet applikasjoner, blir derfor ofte spesifisert i Deployment-ressurser. Dette kan være montering av volumer i form av PersistantVolumes, konfigurasjonsinnstillinger som miljøvariabler, og annen konfigurasjon som oppsett av automatisk skalering og *Healthchecks*. Dette har vi gjort for de forskjellige komponenter på serveren. Ellers ved oppgradering av applikasjoner, benytter man Deployment for å spesifisere ny versjon. Kubernetes vil deretter automatisk rulle ut den nye versjonen til de Pod-ene som er en del av Deploymentet.

3.7.3.7 PersistenVolumeClaimes

For å ikke binne Deploymentene 3.7.3.6 for tett til en spesiell type lagring valgte vi å bruke Kubernetes sitt *PersistenVolumeClaimes*(PVC) system. Med dette systemet kan vi abstrahere bort detaljene om hva den underliggende leverandøren av lagringsplassen er. Alt vi trenger å definere er hvor mye lagringsplass vi ønsker og noen enkle krav til hvem som får lov til å fylle denne forespørselen om lagringsplass.

3.7.3.8 PersistenVolumes

For å kunne fylle PVC'ene 3.7.3.7 brukte vi Kubernetes *persisten volumes* (*Persistent Volumes* 2021). Dette er volum objekter som man kan binde til mange forskjellige lagringsløsninger.

3.7.3.9 Service

Service er en ressurs i Kubernetes som muliggjør at applikasjoner kan kommunisere med hverandre innad i et cluster. Dette muliggjøres ved hjelp av Kubernetes sitt innbygde service discovery-system. Ved å opprette *Service*-ressurser i Kubernetes, vil disse bli lagt til i *service discovery*, og kan nås av alle *Pods* som kjøres i clusteret. Når man definerer en ny *Service*-ressurs, skriver man hvilke porter tjenesten skal lytte på og hvilke Pod-er nettverkstrafikken

skal sendes til.

Når en Service-ressurs bringes opp vil Kubernetes lage en dedikert IP-adresse innad i clusteret kalt *Cluster IP*. Samtidig blir det også laget en DNS-peker mot den opprettede IP-adressen. All trafikk som blir sendt til *Service*-objektet vil bli videresendt til Pod-ene som er definert i spesifikasjonen. Man kan også definere flere containere, og Kubernetes vil last balansere trafikken mellom disse. De opprettede IP-adresser og DNS-pekere blir så til slutt tilgjengelige for de andre containerne i form av *miljøvariabler*.

3.7.3.10 Ingress

For at klienter og eksterne tjenester skal ha mulighet for å kommunisere inn til en mikrotjeneste som kjører i Kubernetes benyttes en Ingress-ressurs (*Ingress* 2021). En analogi til dette kan være sluseport i en kanal, som all båttrafikken benytter for å komme gjennom kanalen. Når nettverks trafikk når Ingress-ressursen, vil den inspiseres og videresendes til rett Service-ressurs for videre prosessering. I praksis utgjør derfor en Ingress-ressurs på mange måter tilsvarende oppgaver som en *Reverse Proxy*-server 2.9.7. Vi har tatt i bruk en Ingress-ressurs, slik at applikasjonsserveren som kjører i Kubernetes kan nås fra Internett. På denne måten kan mobilapplikasjon 4.4 og betalingsleverandør 3.7.5.1 kommunisere med REST API-endepunktene 2.9.4 de bruker.

En ting som er spesielt med Ingress, er at den er todelt. Ingress-ressurser i seg selv kjøres ikke i Kubernetes-clusteret, men heller er rene spesifikasjonsfiler som forteller hvordan en Ingress Controller skal behandle trafikken. En Ingress Controller implementerer funksjonaliteten som utgjør en Ingress, og kjøres utenom Kubernetes-clusteret. Derfor må den installeres av brukeren etter hva som passer i kjøremiljøet der clusteret er installert (Burns 2019, s. 90). For eksempel hvis man kjører clusteret hos en skytjeneste som Amazon Web Services, kan man benytte deres ELB-last balanserer. Siden vi kjører clusteret under utvikling med MicroK8s 3.7.4.3, har vi benyttet *NGINX Ingress Controller* for å gjøre dette.

3.7.3.11 Secrets

Å putte sensitiv informasjon i config filer og deployment filer åpner muligheten for uheldige lekkasjer av denne infoen.

For å effektivt og trygt kunne håndter og distribuere sensitiv informasjon til Pod-ene brukte vi Kubernetes *secrets* (*Secrets* 2021). *Secrets* er et system Kubernetes har hvor en kan definere strenger som lagres som nøkkel-verdi par i clusteret. En pod kan da senere hente disse verdiene ved å referere til nøkkelen i sin spesifikasjons fil.

3.7.3.12 Healthcheck

For å kunne vite om containerne i applikasjonen har startet opp og kjørte måtte vi bruke Kubernetes sitt system for tilstandssjekk (*Configure Liveness, Readiness and Startup Probes* 2021). Med dette systemet kan vi definere hvilke endepunkt som skal spørres om tilstand og hvor mange forsøk en pod har på å svare.

3.7.3.13 HorizontalPodAutoScaler

En funksjon i Kubernetes som gruppen ønsket å prøve ut var automatisk skalering av containerne basert på last. Dette kan gjøres gjennom Kubernetes *HorizontalPodAutoScaler* (*Horizontal Pod Autoscaler* 2021). Med denne kan man definere forskjellige kriterier for skalering og de-skalering.

3.7.3.14 cert-manager

Siden gruppen ønsket kjøre all trafikk på HTTP/TLS av sikkerhets grunnlag måtte vi ha et system for å håndtere TLS-nøkler og HTTP sertifikat3.7.5.2. Dette kunne ikke gjøres med de inkluderte verktøyene i Kubernetes. Dette er fordi at ingress provideren også skalerer og vil derfor fort få problemer med å svare på utfordringen til sertifikat utgiveren. Dette er fordi det ikke er noen garanti på at svaret på utfordringen kommer tilbake til rett ingress-provider. Løsningen var å bruke *Cert-manager* (*Welcome to cert-manager* 2021). Cert-manager vil ta seg av å håndtere utfordringene gitt fra sertifikat utgiverne, og vil lagre sertifikatet i en Ku-

bernetes secret 3.7.3.11 som de forskjellige ingressene kan hente inn og bruke. Cert-manager vil også følge med at sertifikatene ikke er utgått og fornye de når utgangsdato nærmer seg.

3.7.4 Teknologier

For å sette opp serverens kjøremiljø hadde vi behov for å bruke et sett med teknologier. Her forklarer vi hva de gjør.

3.7.4.1 Apache Maven

Maven er et byggesystem som brukes i Java, og det automatiserer håndtering av tredjeparts-biblioteker. Når et Java-prosjekt bruker Maven, representeres prosjektet og dets biblioteker i det som kalles en POM-modell, på engelsk kjent som *Project Object Modell*. POM-modellen lagres i prosjektet sin filstruktur som XML-filer med navn *pom.xml*. Et Java-prosjekt må minst ha en slik fil for å kunne bygges med Maven. Flere *pom.xml*-filer kan peke til hverandre, og de kan hente innhold fra hverandre. Dette er kjent som et *multimoduls-prosjekt*, og har blitt benyttet som prosjektstruktur for applikasjonsserveren 4.3.

3.7.4.2 Docker

Docker er et sett av "platform as a service (PAAS)" produkter som bruker OS-nivå virtualisering 2.8.3 for å utgi programvare i pakker kalt containere 2.8.3.1. Containere er isolert fra hverandre og inneholder deres egen programvare, biblioteker og konfigurasjons filer. De kan kommunisere med hverandre gjennom definerte kanaler (Wikipedia 2021e). Fordi alle containerne deler enkel operativ system"kernel" bruker de mindre ressurser enn virtuelle maskiner. Dette gjør at en enkel server eller virtuell maskin kan kjøre flere containere samtidig. Vi valgte å bruke Docker fordi det gjør det enkelt å kjøre "backend"server og database på en fysisk maskin. Dette gjør det også enklere å raskt sette alt opp for å teste på egen maskin.

3.7.4.3 MicroK8s

For å ha mulighet for å kjøre Kubernetes lokalt på våre egne maskiner tok vi i bruk verktøyet MicroK8s. Verktøyet er utviklet av Canonical, og har støtte for både Mac, Windows og Linux. MicroK8s installerer et komplett Kubernetes driftsmiljø, uten at en trenger spesialisert kunnskap på hvordan dette gjøres (Canonical 2021). Det blir også installert vanlige tilleggskomponenter, som man normalt må installere manuelt. På denne måten fikk vi raskt satt opp lokalt Kubernetes-cluster på hver våre maskiner. Dette har vært medvirkende for at vi kunne fortsette å utvikle applikasjonsserveren på våre egne maskiner, selv etter at vi gikk over til å bruke Kubernetes.

3.7.4.4 Payara

Siden serverkoden i dette prosjektet er skrevet som en JakartaEE-applikasjon, er det behov for å bruke en applikasjonsserver som støtter JakartaEE spesifikasjonene. For å kjøre serveren som en monolittisk applikasjon, har vi benyttet Payara Server.

Payara Server er en applikasjonsserver utviklet av Payara Services Ltd, som en videreutvikling av applikasjonsserveren GlassFish, opprinnelig lansert av Sun Microsystems i 2005. Payara og GlassFish er utviklet som fri programvare og tilbys kostnadsfritt. Payara Server ble først utgitt i 2010 for å tilby et alternativ for kommersiell støtte av GlassFish, da Oracle som drev utviklingen av GlassFish bestemte seg for å avslutte muligheten for dette. Payara Server har etter dette vært tilgjengelig i to utgaver; Payara Community Edition som er kostnadsfri, og Payara Enterprise Edition som selges på lisens med brukerstøtte. Det ble først bestemt å bruke Payara Community Edition fordi produktet har aktiv utvikling, støtter JakartaEE og MicroProfile, fungerer godt med Docker og har mulighet for kommersiell støtte.

3.7.4.5 IBM OpenLiberty

Underveis i prosjektet gjorde vi om serveren fra en monolitt til en mikrotjeneste. For å kjøre serverkoden som en mikrotjeneste i Kubernetes, har vi brukt applikasjonsserveren OpenLiberty. Som Payara, har også OpenLiberty støtte for JakartaEE-standardene og MicroProfile. Produktet er basert på WebSphere Liberty, som er IBM sin kommersielle løsning for å drive

JakartaEE-applikasjoner (IBM 2017). OpenLiberty i seg selv utvikles av IBM som åpen kildekode, og kan brukes uten kostnad. Dokumentasjonen til OpenLiberty er oversiktlig, og det er flere eksempler tilgjengelig på hvordan man implementerer funksjonalitet som er vanlig å benytte i mikrotjenester.

Funksjonalitet i OpenLiberty kan aktiveres eller deaktiveres etter behov, slik at bare nødvendig funksjonalitet kjøres (IBM 2021d). Dette er til stor forskjell fra Payara 3.7.4.4, som implementerer komplett funksjonalitet for et JakartaEE-rammeverk. Det er både fordeler og ulemper med et funksjons-orientert Feature-system. Ved å bare kjøre funksjoner som er absolutt nødvendige, vil dette korte ned oppstartstiden betraktelig og gjøre at serveren bruker mindre ressurser. Men det kan også føre til frustrasjon, for en kan komme borti å bruke mye tid på feilsøking viss man benytter funksjonalitet fra Features som ikke er aktivert. Derfor må man være påpasselig med å sjekke at API-er man benytter, er aktiverte. For hver av JakartaEE-spsesifikasjonene vi benyttet, har vi aktivert OpenLiberty Features som implementerer støtte for dette. Vi forteller mer om dette i detalj under Del 3.14.1.

3.7.4.6 EclipseLink

EclipseLink er et ORM-rammeverk 2.13.1.1 som springer ut i fra prosjektet *Eclipse Persistance Services Project* fra Eclipse Foundation. Rammeverket implementerer programmeringsgrensesnittet Java Persistance API, som er en del av spesifikasjonene i JakartaEE-rammeverket. EclipseLink tilbyr funksjonalitet mot konvensjonelle relasjonsdatabaser som SQL, men også mot andre databaser som bruker modeller som avviker fra relasjonsmodellen, også kjent som NoSQL-databaser (Eclipse Foundation 2014).

3.7.4.7 PostgreSQL

Når serveren både har kjørt som monolitt og som mikrotjeneste, har data som skal skrives til database blitt lagret med PostgreSQL. PostgreSQL er et databasesystem DBMS som brukes til å opprette og styre relasjonsdatabaser 2.13.1 basert på SQL-standarden. Systemet har sine røtter tilbake til 1986, som en del av oppstarten til *The Berkley POSTGRES Project* ved Universitetet i California, Berkley. PostgreSQL har siden utviklet seg til et av verdens mest utbredte SQL-baserte relasjonsdatabaser, og er tilgjengelig som fri og åpen programvare. Database-

systemet implementerer i stor grad siste versjon av SQL-standarden; PostgreSQL 13 som ble lansert høsten 2020 implementerer 170 av 179 krav fra SQL:2016 (PostgreSQL 2021). Det ble besluttet å ta i bruk PostgreSQL fordi den er fritt tilgjengelig, har støtte for Docker, og at medlemmene i prosjektet har god erfaring med bruk av databasesystemet fra tidligere prosjekter.

3.7.4.8 JSON Web Token

For å autentisere brukere etter de har logget inn brukte vi JSON Web Token (JWT) som er en form for token-basert autentisering 2.11.2. JWT er en standard som definerer en kompakt, URL-kompatibel metode for å representere informasjon som skal sendes mellom to parter. Informasjonen i JWT er kodet som et JSON objekt (Jones, Bradley og Sakimura 2015, s. 6). For å være sikker, blir en JWT signert av serveren med en privat nøkkel. Da når JWT-en blir sendt til serveren som autentisering, kan serveren verifisere at det er autentisk.

3.7.5 Eksterne API-er

For å kunne ta i bruk en del funksjonalitet, benytter vi oss av kommunikasjon mot API endepunkter som tilbyes av andre aktører på Internett. Under forklarer vi i korte trekk hvilke aktører og hvorfor det kommuniseres med dem.

3.7.5.1 Nets EasyAPI

Easy API leveres av Nets, og er en løsning som gjør det mulig å tilby kortbetaling inne i mobilapplikasjonen. Selve transaksjonen foregår på en nettside driftet hos Nets, som vi får en lenke til når vi registrerer en ordre via Easy API.

3.7.5.2 Let's Encrypt

Let's Encrypt er en tjeneste som tilbyr utstedelse av TLS-sertifikater for domenenavn uten kostnad (Internet Security Research Group 2021). Tjenesten er ikke-kommersiell, er svært utbredt og har høy tillit i IT-bransjen. Utsendelse av sertifikater foregår automatisk ved hjelp

av en egen protokoll, og mange teknologier har implementert støtte for Let's Encrypt. Vi benytter Let's Encrypt sitt API for å få tildelt sertifikater ved hjelp av *cert-manager* 3.7.3.14.

3.8 Mobilpplikasjon

Hovedmålet i prosjektet var og utvikle en mobilapplikasjon, her beskriver vi hvilke teknologier, rammeverk og metoder vi brukte for å kunne få til dette.

3.8.1 Teknologier

Et av kravene i kravspesifikasjon fra oppdragsgiver var at applikasjonen måtte kunne kjøres både på plattformene iOS og Android. Tradisjonelt sett ville dette kreve utvikling av separate kodebaser for hver av plattformene, som ville ha vært ressurskrevende tidsmessig. Vi så derfor tidlig behov for å ta i bruk et kryssplattformrammeverk 2.10 for å understøtte utviklingen.

3.8.1.1 Vurderte rammeverk

De mest utbredte plattformer for kryssplatformutvikling i dag er Xamrin basert på C#, React Native basert på JavaScript og Flutter basert på Dart. Xamarin ble tidlig valgt bort da det bare støtter Windows og MacOS-plattformer, mens to av gruppemedlemmene bruker Linux-baserte operativsystem. React Native ble også vurdert, men ansett som uaktuelt fordi ingen av medlemmene i gruppen hadde arbeidet med JavaScript i noen stor grad tidligere. En av gruppemedlemmene var derimot allerede kjent med Flutter fra før, og etter litt utprøving innad i gruppen ble det besluttet å ta i bruk rammeverket.

3.8.1.2 Flutter

Flutter er et kryssplattformrammeverk 2.10 utviklet av Google (*Flutter (software)* 2021), for å muliggjøre utvikling av applikasjoner mot flere plattformer fra en enkelt kodebase skrevet i Dart 3.10.2. En av grunnene til at vi valgte Flutter var at Java, som vi var kjent med fra før, ikke støtter kryssplatformutvikling for mobilapplikasjoner. Hadde vi valgt å bruke Java hadde vi

blitt nødt til å lage to applikasjoner: en for Android med Java, og en for iOS med Swift eller et annet språk.

3.9 Webapplikasjon

For å ha et system for å administrere brukere på plattformen, valgte vi å lage en webapplikasjon. Den skulle fungere som et grensesnitt for administratorer å administrere brukere og varekategorier. For å lage denne webapplikasjonen valgte vi å bruke React.js.

3.9.1 React.js

React.js er et "open source" JavaScript 3.10.3 bibliotek som ble lansert av Facebook i mai 2013 (Thakkar 2020, s. 41). Det blir brukt for å lage brukergrensesnitt (UI). Hoveddelen av React er komponenter. De ligner på funksjoner i at de tar inn en input, gir tilbake UI elementer, og kan brukes når de trengs i forskjellige filer. Når data som brukes av React endres, endres også UI elementene som bruker denne data-en (Thakkar 2020, s. 45–46).

3.10 Programmeringsspråk

Et programmeringsspråk er et formelt språk bestående av et sett av instrukser som produserer forskjellige typer "output". Programmeringsspråk blir brukt i datamaskin programmering for å implementere algoritmer.
I dette prosjektet ble det brukt forskjellige programmeringsspråk ved utvikling av mobilapplikasjon, server og webapplikasjon.

3.10.1 Java

Vi valgte å bruke programmeringsspråket Java for å lage vår "backend" fordi vi alle på gruppen er godt kjent språket fra tidligere og vi har brukt det ved lignede prosjekter før. Java er et klasse-basert, objekt-orientert språk utviklet ved Sun Microsystems og lansert i 1995. Det er

et generelt programmeringsspråk ment å la applikasjonsutviklere *write once, run anywhere*. Dette betyr at kompilert Java-kode kan kjøre på alle plattformer som kan kjøre *Java Virtual Machine (JVM)* uten at koden må rekompileres. Java kompileres ikke til maskinkode, men til plattformuavhengig bytekode som kan kjøres på enhver JVM uavhengig av den underliggende maskinvaren (Wikipedia 2021g).

3.10.2 Dart

Ved utvikling av mobilapplikasjonen brukte vi rammeverket Flutter, som baserer seg på programmeringsspråket Dart. Dart er et klient-optimalisert språk for applikasjoner på flere plattformer. Det er utviklet av Google og blir brukt til å lage applikasjoner for smarttelefoner, "desktop", server, og web. Det er et klasse-basert, objekt-orientert språk med syntax som ligner på språket C. Dart kan bli kompilert til enten maskinkode eller JavaScript (Wikipedia 2021c).

3.10.3 JavaScript

Vi har benyttet JavaScript, fordi webapplikasjonen i prosjektet ble laget med React 3.9.1. JavaScript er et programmeringsspråk som benyttes for utvikling av webapplikasjoner. Språket er standardisert som ECMAScript, og forvaltes av Ecma International (Mozilla Developer Network 2021). Tradisjonelt sett ble språket laget for å kjøres i nettlesere, men benyttes i dag også for å lage server-sideapplikasjoner og mobilapplikasjoner. Det er et "scripting-språk", det vil si at programkoden ikke kompileres, men heller tolkes av kjøremiljøet etter hvert som programmet utføres. Misvisende nok har JavaScript og Java nesten like navn, men de er ikke relaterte utenom at JavaScript deler litt av syntaksen til Java (Nätt 2021).

3.10.4 TypeScript

For å ha støtte for statiske typer når vi benyttet JavaScript 3.10.3, tok vi også i bruk TypeScript. TypeScript er et programmeringsspråk som implementerer støtte for statiske typer i JavaScript. Språket utvikles av Microsoft, og er et super sett av JavaScript. Det vil si at all kode

skrevet som JavaScript er også gyldig TypeScript. Fordi JavaScript er et tolket språk, vil ikke typefeil oppdages før koden kjøres. TypeScript tilnærmer seg å løse dette problemet, ved å tilby støtte for å definere statiske typer i kildekoden. Disse statiske typene muliggjør at feil kan avdekkes, før koden blir kjørt. Ved utvikling med TypeScript *kompileres* ikke koden, men oversettes heller til standard JavaScript. Denne oversettingen kalles *transpilering* men må ikke forveksles med *kompilering*, fordi koden er lesbar både før og etter prosessen. (Microsoft 2021b).

3.11 Testmetodikk

For å enkelt finne ut om koden fortsatt fungerer som den skal etter at vi har endret på den, har vi tatt i bruk automatiserte tester. For å teste enkelte små enheter på serveren brukte vi Unit-testing.

3.11.1 Unit Testing

Unit testing 2.5.2.1 er en testmetodikk som vi brukte. For å kunne unit teste i Java trenger man noen eksterne rammeverk.

3.11.1.1 JUnit

JUnit er et rammeverk for unit testing i Java. Vi valgte å bruke JUnit fordi det er et rammeverk alle på gruppen er kjent med fra før, noe som gjør det raskere å skrive tester.

3.11.1.2 Mockito

Mockito er et rammeverk for Java som gjør det mulig å lage “mock”-objekter i unit tester.

3.11.1.3 Weld-junit

Weld-junit er et test rammeverk for kunne overstyre hva som blir injisert i fra Weld (*Weld home page* 2021) som er serverens implementasjon av JakartaEE CDI 3.7.1.4. Dette er veldig nyttig når man skal unit-teste serveren.

3.12 Designredskaper

Her beskriver vi hvilke redskaper vi brukte for å kommunisere design med oppdragsgiver.

3.12.1 Adobe XD

Adobe XD er et verktøy utviklet av Adobe Inc. (Wikipedia 2021a), for å designe brukergrensesnitt og prototyper for nettsider, applikasjoner og andre digitale produkter. Programmet har et stort bibliotek av innebygde designelementer og skjermutforminger fra plattformer som iOS og Android. Mange utgivere av designmaler, som for eksempel Google ved Material Design (*Material Design* 2021) utgir filer man kan laste ned og importere. Dette gjør at designet i Adobe XD enkelt kan implementeres i kode med tilsvarende elementer, og sluttresultatet kan se tilnærmet identisk ut mellom design og ferdig produkt. Et design kan eksporteres til vanlige bildeformater, men også web - nyttig ved behov for å dele designet med eksterne, som oppdragsgiver. Programmet finnes i en gratisversjon som baserer seg på lokal lagring, og en betalversjon (Adobe 2021) som støtter synkronisering og samarbeid via Adobe sin skyløsning. I et prosjekt med såpass få involverte som tre personer kan en synkronisere designfilen manuelt, og basere seg på å bruke gratisversjonen. På grunn av at programmet hadde tilstrekkelig funksjonalitet, og kunne brukes kostnadsfritt - ble det bestemt å bruke Adobe XD for design av brukergrensesnittet for mobilapplikasjonen.

3.13 Utviklingsverktøy

Underveis i utviklingen brukte vi en sett med forskjellige verktøy til oppgaver som å skrive kode og testing av API. Her beskriver vi hvordan vi brukte disse verktøyene.

3.13.1 JetBrains Intellij IDEA

IntelliJ IDEA et integrert utviklings miljø (IDE) som brukes for å utvikle programvare. IntelliJ har to versjoner, en open source "community" versjon, og en proprietær kommersiell versjon. IntelliJ tilbyr kode assistanse gjennom egenskaper som kode fullføring ved å analysere konteksten, "code refactoring", og "code debugging" (Wikipedia 2021f). IntelliJ støtter "plugins" som man kan bruke for å legge til funksjonalitet. IDE-en støttet originalt kun Java, men har etter hvert fått støtte for mange programmeringsspråk, de fleste gjennom "plugins". Vi valgte å bruke IntelliJ, fordi det er en IDE med mye funksjonalitet og den kan få utvidet funksjonalitet via plugins. Vi har også tilgang til den kommersielle versjonen via NTNU.

3.13.2 Android Studio

Android Studio er den offisielle IDE-en for Google's Android operativsystem. Det er bygget på IntelliJ (3.13.1) og er designet spesifikk for Android-utvikling. Android Studio er "open source" og har de fleste funksjonene til IntelliJ. I tillegg har Android Studio egne funksjoner rettet mot Android utvikling. IDE-en har blant annet veivisere for å lage vanlige Android design og komponenter fra maler. Det er også innebygget en emulator (Android Virtual Device) for å kjøre og utføre debugging av applikasjoner (Wikipedia 2021b). Vi valgte å bruke Android Studio fordi den er veldig lik IntelliJ, og mye støtte for Android-utvikling.

3.13.2.1 Android Virtual Device

Som nevnt i seksjon 3.13.2, så har Android Studio en funksjon kalt Android Virtual Device (AVD). Dette er en emulator for å etterligne en fysisk enhet med Android. Gjennom AVD har

man mulighet til å velge flere typer maskinvare man vil kjøre Android på, og flere versjoner av Android (Google 2021). Dette kan brukes for å teste og "debugge" en applikasjon.

3.13.3 Insomnia

Insomnia er en API-klient utviklet av Kong Inc. Programmet er utviklet som åpen kildekode og støtter flere protokoller som REST, SOAP, GraphQL, og gRPC. I Insomnia kan man opprette API-forespørsler og organisere de. Man kan også sette dynamiske variabler for å håndtere flere miljøer som utvikling og produksjon. Insomnia lar en også generere kode ut fra API-forespørslene man har laget (Kong Inc 2021). Insomnia er gratis å bruke, men det har abonnement som gir tilgang til blant annet synkronisering av API- forespørsler for enkeltpersoner og team. Vi valgte å bruke Insomnia fordi det lar oss eksportere API- forespørslene som en YAML fil slik at vi kan dele den med hverandre og ha versjons kontroll på den.

3.13.4 WSL2

Windows Subsystem for Linux 2 er et kompatibilitetslag som gjennom en underdel av Hyper-V funksjoner kjører en Linux kernel i en VM. Første versjon av WSL gir tilgang til et Linux-kompatibelt kernel grensesnitt som inneholder ingen Linux-kernel kode. Dette lar deg kjøre et GNU Bash "shell" og kommando språk (Wikipedia 2021l). WSL2 endret på den underliggende arkitekturen til å kjøre en komplett Linux kernel i et lettvekt VM miljø. Dette tillater at Linux programmer som må gjøre systemkall nå kan fungere, slik som Docker (3.7.4.2) (Microsoft 2020). Vi valgte å bruke WSL2 fordi noen av oss bruker Windows og andre Linux. WSL2 gjør det da enkelt at alle kjører utviklingen i samme miljø.

3.14 Eksterne biblioteker

For å kunne fokusere på den faktiske problemstillingen, har vi benyttet eksterne biblioteker underveis i utviklingen. Biblioteker er programkode som gjør spesifikke oppgaver svært godt, men tilgjengeliggjør disse oppgavene for andre utviklere på en enkel måte. For populære programmeringsspråk finnes det ofte mange biblioteker, som implementerer utbredte

og vanlige standardiserte funksjoner. Dette kan for eksempel være kommunikasjonsprotokoller, der det normalt kreves svært spesifikk kunnskap om hvordan protokollen fungerer. Ved hjelp av et bibliotek som implementer en slik protokoll, kan utvikleren heller fokusere på applikasjonen vedkommende lager. Vi har benyttet eksterne biblioteker både på mobilapplikasjonen, applikasjonsserveren og webapplikasjonen. I Del 3.14.1, 3.14.2 og 3.14.3 forteller vi hvilke biblioteker vi har benyttet.

3.14.1 Applikasjonsserver

OpenLiberty 3.7.4.5 bruker definisjonen *Features* som deler opp forskjellig funksjonalitet for applikasjonsserveren i moduler. En *feature* aktiverer spesifikk funksjonalitet, på lik måte som man importerer et eksternt bibliotek i en vanlig applikasjon. *Features* gjøres i filen *server.xml*, som styrer konfigurasjonen for OpenLiberty. Nedenfor forteller vi om hvilke *Features* vi har benyttet oss av. I tillegg har vi også benyttet det eksterne biblioteket Lombok når vi har utviklet i Java.

3.14.1.1 microProfile-4.0

microProfile-4.0 implementerer støtte for MicroProfile-spesifikasjonen 3.7.2. Denne er aktivert fordi vi benytter oss av funksjonalitet fra MicroProfile i koden for serveren.

3.14.1.2 jakartaee-8.0

For at vi skal kunne bruke de forskjellige spesifikasjonene vi har programmert mot i Jakarta-EE 3.7.1, måtte denne *Featuren* aktiveres på serveren. Når denne er aktivert, vil den aktivere alle av OpenLiberty sine *Features* som implementerer de forskjellige modulene som inngår i JakartaEE.

3.14.1.3 Lombok

Lombok er et bibliotek som har hjulpet oss med å med å forenkle utvikling av Java-koden på serveren. Lombok kan automatisk generere kode for vanlige metoder i Java. Dette gjøres ved

hjelp av *Java-annoteringer*.På denne måten består koden av færre linjer, noe som gjør den mer lesbar for oss selv og andre. Vi har brukt *@Data, @Getter, @Setter*, samt annoteringer for å lage konstruktørene. Under er et utvalg av de mest vanlige annoteringene som benyttes i Lombok:

- **@Getter**
  Felt som er annotert med denne, får generert en Get-metode som kan brukes for å lese data.
- **@Setter**
  Genererer Set-metoder som kan brukes av andre klasser for å skrive data.
- **@Data**
  Denne lager følgende på en klasse: @Getter metoder for alle felt, @Setter for alle felt som ikke er *final*, *toString* for felt og @RequiredArgsConstructor.
- **@NoArgsConstructor**
  Lager en konstruktør for klassen uten parametre.
- **@RequiredArgsConstructor**
  Lager en konstruktør med parametre for felt som er annotert med @NonNull eller *final*.
- **@AllArgsConstructor**
  Lager en konstruktør som inneholder parametre for alle felt i klassen.

Nedenfor viser vi hvordan Lombok genererer kode for *@Getter, @Setter* og *@NoArgsConstructor*. Kodeeksempel 3.1 viser kildekoden med *Java-annoteringer* og Kodeeksempel 3.2 viser kode som Lombok lager. Etter kompilering vil metoder fra begge kodeeksempler være inkludert i klassen.

Kodeeksempel 3.1: Lombok-kildekode

```
1  @NoArgsConstructor
2  public class FishingVessel {
3      ...
4      private long registrationNumber;
5      ...
6      @Getter
7      private long registrationYear;
8      ...
9      @Setter
10     private String name;
11 }
```

Kodeeksempel 3.2: Genereres av Lombok

```
1  public class FishingVessel {
2      public FishingVessel() {
3      }
4      public long getRegistrationYear() {
5          return registrationYear;
6      }
7      public void setName(final String
           name) {
8          this.name = name;
9      }
10 }
```

3.14.2 Mobilapplikasjon

Flutter bruker programmeringsspråket Dart, og i dette prosjektet har vi benyttet oss av tredjepartsbiblioteker for å dra nytte av utvidet funksjonalitet. En viss risiko er det ved å ta i bruk tredjepartsbiblioteker, og for hvert bibliotek må man nøye vurdere nytte og konsekvenser. Heldigvis for oss har Dart en god nettside som samler alle bibliotek-pakker for språket.

Registrerte brukere kan på siden publisere sine egne pakker og etter kvalitet blir de rangerte med poeng. Dette har forenklet arbeidet med å vurdere og velge ut passende bibliotekpakker. Tidsbesparelsen en får med å ta i bruk gode bibliotek-pakker kontra utvikle slik funksjonalitet selv, gjør det mulig å realisere en applikasjon over kortere tid.

3.14.2.1 http

For å sende informasjon til og fra serveren fra mobilapplikasjonen bruker vi HTTP 2.9.3. Biblioteket *http* inneholder funksjoner og klasser som gjør det enkelt å sende og motta HTTP resurser.

3.14.2.2 json_annotation

Vi sender JSON objekter mellom serveren og mobilapplikasjonen og trenger derfor en metode for å gå mellom Dart objekter og JSON objekter. Dette gjør vi ved hjelp av *json_annotation* og *json_serializable*. *Json_annotation* er et bibliotek som brukes for å definere merknadene som brukes av *json_serializable* for å lage kode for JSON serialisering og de-serialisering.

3.14.2.3 json_serializable

Json_serializable gir tilgang til konstruktører for å håndtere JSON. Konstruktørene genererer kode når de finner medlemmer som er merket med klasser definert i *json_annotation*.

3.14.2.4 strings

Strings er et bibliotek med hjelpefunksjoner for strenger. Det kan for eksempel gi ord stor forbokstav eller "camelize" (f.eks DetteErCamelize).

3.14.2.5 map_launcher

Map_launcher er et Flutter bibliotek for å finne installerte kart-applikasjoner på en enhet. Den kan starte disse applikasjonene med en markør eller veivisning.

3.14.2.6 flutter_svg

For å kunne vise ikonene til kart-applikasjonene som vises med *map_launcher* trenger vi å kunne tegne SVG filer. *Flutter_svg* gir oss denne muligheten.

3.14.2.7 flutter_map

Flutter_map lar oss vise et interaktivt kart i applikasjonen vår. Biblioteket har støtte for å vise kart fra OpenStreetMap og Microsofts Azure Maps. Vi valgte å bruke *Flutter_map* istedet for

Google Maps fordi det har all funksjonaliteten som vi trenger, og det lar oss bruke OpenStreetMap som er "open source" og fritt tilgjengelig.

3.14.2.8 geolocator

Geolocator er et bibliotek som gir enkel tilgang til plattform spesifikke lokasjonstjenester. Det lar oss hente f.eks. nåværende posisjon og sist kjente posisjon. Biblioteket lar oss også kalkulere avstanden mellom to koordinater.

3.14.2.9 provider

Biblioteket *provider* forenkler håndtering av tilstand i Flutter-applikasjoner. Biblioteket gjør det mulig å definere egne klasser som arver fra *ChangeNotifier*, som er en observerbar klasse. En observerbar klasse kan overvåke og agere på endringer som skjer i tilstand på et objekt. Klasser som i applikasjonen har behov for å lese tilstand, kan gjøre dette med notasjonen *Provider.of<KlasseNavnMedTilstand>*. Tradisjonelt har man gjerne benyttet tilbakekallsprosesser (callback) for dette. Ved å bruke *provider* for å håndtere tilstand, er dette mer fleksibelt enn *callbacks* og forenkler koden vesentlig. Vi har benyttet biblioteket for applikasjonsdata og håndtering av samtaledata i Chat-delen 4.4.2.15.

3.14.2.10 dropdown_search

Dropdown_search er et bibliotek som lar oss lage nedtrekks-menyer med søkefunksjon. Biblioteket lar oss enkelt tilpasse menyen slik at den passer inn med resten av elementene i applikasjonen.

3.14.2.11 flutter_chat_bubble

For chat funksjonaliteten i applikasjonen ønsker vi at chat meldingene dukker opp i bobler for å differensiere mellom meldingene. Til dette bruker vi *flutter_chat_bubble*. Dette biblioteket lar oss lage forskjellige typer chat-bobler.

3.14.2.12 flutter_secure_storage

Vi har behov for å lagre JWT 3.7.4.8 på en sikker måte på enheten som kjører applikasjonen, og derfor bruker vi *flutter_secure_storage*. Flutter_secure_storage lagrer data på sikre måter på enheter ved å bruke *Keychain* på iOS og *KeyStore* på Android. På Android blir dataen først kryptert med AES (Advanced Encryption Standard), AES nøkkelen blir kryptert med RSA (Rivest–Shamir–Adleman), og RSA nøkkelen blir lagret i KeyStore.

3.14.2.13 shared_preferences

For å kunne lagre en innlogget bruker på enheten som kjører applikasjonen bruker vi *shared_preferences*. Dette biblioteket pakker inn plattform-spesifikk vedvarende lagring av enkel data og vi trenger derfor ikke tenke på om applikasjonen kjører på iOS eller Android.

3.14.2.14 cupertino_icons

Cupertino_icons er en pakke som inneholder standardsettet med ikoner som brukes av Flutter's Cupertino widgets. Inneholder mange ikoner som vi bruker i applikasjonen.

3.14.2.15 intl

Vi har behov for å ha applikasjonen både på norsk og engelsk. Derfor bruker vi biblioteket *intl*. *Intl* gir muligheten for internasjonalisering og lokalisering.

3.14.2.16 intl_utils

Intl_utils er ett bibliotek som lager en binding mellom oversettelser fra *.arb*-filer og en Flutter applikasjon. *Intl_utils* generer lokaliseringskode fra *.arb*-filer. ARB-filer er filer hvor vi skriver inn alle ord og setninger som oversettes i applikasjonen.

3.14.2.17 url_launcher

I applikasjonen har vi URL-er som skal være mulig å åpne enkelt i en nettleser ved å trykke på dem. *Url_launcher* gir oss muligheten for dette. *Url_launcher* er et bibliotek for å åpne en URL i en ekstern nettleser eller i ett *WebView*.

3.14.2.18 string_validator

For å slippe å selv lage mange hjelpemetoder for strenger brukte vi biblioteket *String_validator*. Dette biblioteket inneholder mange hjelpemetoder for strenger slik som *isDate()* som lar oss sjekke om en streng er en dato, og *toInt()* som lar oss konvertere en streng til et heltall.

3.14.3 Webapplikasjon

Siden det finnes millioner av moduler og pakker i javascript (*Modulecounts* 2021) valgte vi å holde det enkelt og kun bruke 3 av disse eksklusive selve React.js.

3.14.3.1 axios

For å kunne enklere gjennomføre HTTP forespørslene til de forskjellige mikrotjenestene fra admin siden valgte vi å bruke Axios biblioteket (*Axios* 2021). Axios tilbyr en løfte basert HTTP-klient for JavaScript.

3.14.3.2 React router

For å kunne navigere mellom de forskjellige sidene i webgrensenittet som vi lagde i React, brukte vi *React router*-biblioteket (*React Router: Declarative Routing for React* 2020). Dette er et bibliotek som tillater oss å utføre navigasjon på en “singel page” nettside.

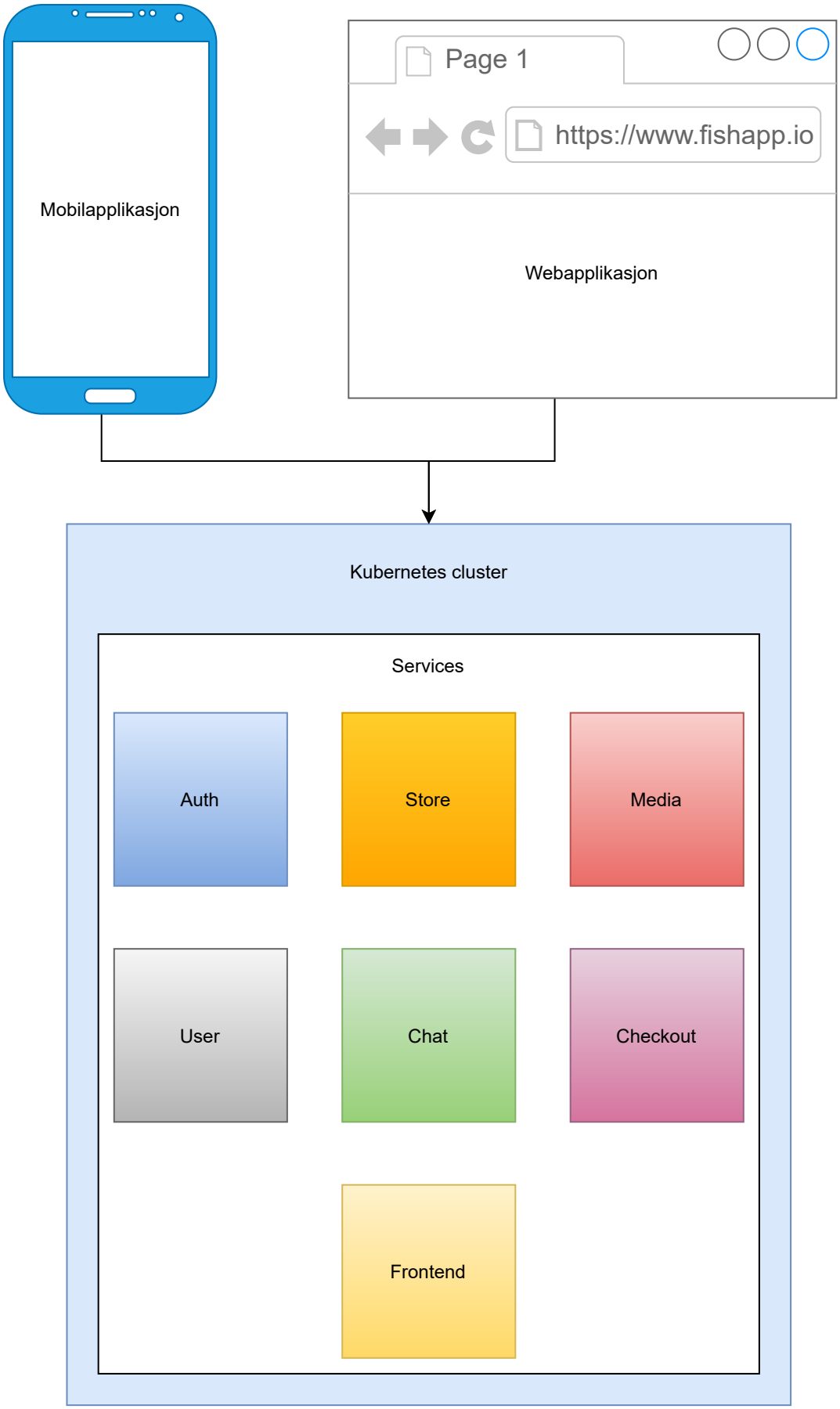
Del 4

Resultater

I denne delen forteller vi om resultatene vi har oppnådd i prosjektet. Her går vi også igjennom hvordan metodene beskrevet i Del 3 ble implementert. Vi forklarer først løsningen i sin helhet med tilhørende diagrammer, og deretter hver del i mer detalj.

4.1 Løsningen i sin helhet

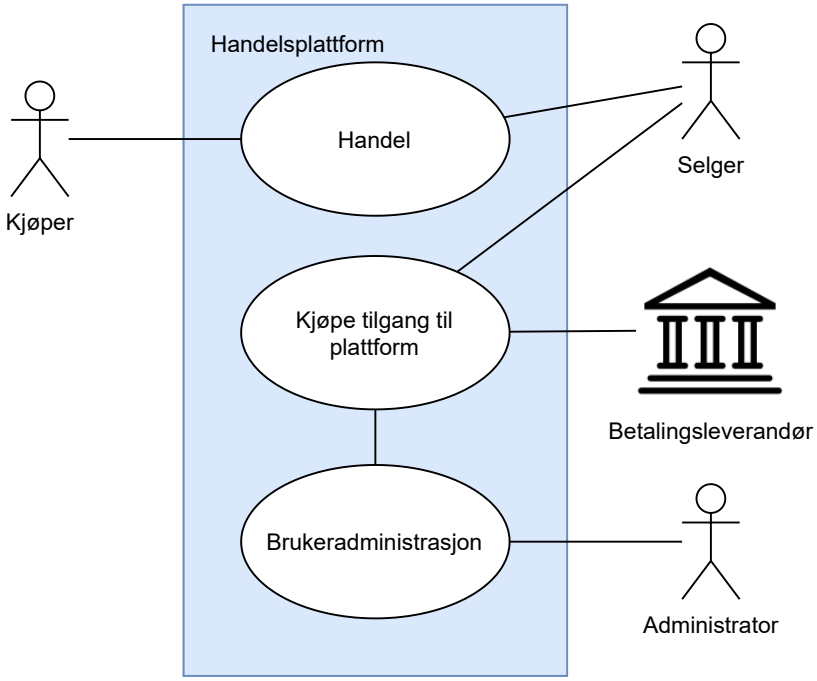
Løsningen i sin helhet innebærer å utvikle en komplett handelsplattform, som skal forenkle handel av sjømat mellom fiskere og sluttbrukere. Plattformen består av tre deler: en applikasjonsserver driftet av oppdragsgiver, en mobilapplikasjon installert hos brukerne, og et webgrensesnitt for administrasjon av tjenesten. Selve kjernen i plattformen er applikasjonsserveren, som bindeledd mellom administrasjonssiden og mobilapplikasjonen. Applikasjonsserveren er bygget på en mikrotjeneste-basert arkitektur, der flere selvstendige komponenter sammen utgjør selve serveren. Figur 4.1 viser et overordnet bilde av løsningen.



Figur 4.1: Diagram av overordnet løsning

4.2 UML - Use Case Diagram

Use Case Diagrammet viser de forskjellige aktørene som systemet samhandler med. Samlet sett er det fire aktører som kommuniserer med systemet. Tre av fire er fysiske personer, og siste aktør er betalingsleverandør sitt system. Figur 4.2 viser de ulike aktørene, og vi forteller mer om disse i sine respektive overskriver nedenfor.



Figur 4.2: Use Case Diagram over systemet i sin helhet

4.2.1 Kjøpere

Kjøpere interagerer med systemet i forkant av og underveis i en handel. Først installerer kjøperen mobilapplikasjonen på sin enhet. Kjøperen kan så se hvilke varer og salgsannonser som er tilgjengelige på handelsplattformen. For å kunne kjøpe en vare, eller kontakte en selger, må kjøper oppette brukerkonto og logge inn med denne. Som innlogget kan kjøper velge ønsket salgsannonse, og starte en samtale med selger eller utføre et kjøp direkte på en annonse. Handelen forblir direkte mellom aktørene kjøper og selger. Administrator som er plattformdrifter, har ikke noe med handelen å gjøre økonomisk sett.

4.2.2 Selgere

For å benytte plattformen som selger, må vedkommende på lik linje som kjøpere ha installert mobilapplikasjonen på sin enhet. Videre må en selger ha opprettet en egen selgerkonto og logget inn på denne. For å få legge ut salgsannonser, må selger kjøpe tilgang. Dette gjøres i form av et abonnement som trekkes månedlig, og skjer mellom aktørene betalingsleverandør og selger. Når selger har gyldig abonnement, vil vedkommende få mulighet for å opprette annonser og utføre handel med kjøperne.

4.2.3 Administratorer

Eierene som drifter handelsplattformen, vil til enhver tid ha mulighet for å administrere den ved at de kan logge inn som Administratorer. Dette gjøres mot eget webgrensesnitt, som kjøres på server. Eierne kan administrere tilgjengelige varetyper, brukere som har tilgang etc.

4.2.4 Betalingsleverandør

Betalingsleverandøren foretar månedlige trekk fra selgere som har kjøpt tilgang til plattformen. Beløp innkrevet av betalingsleverandør overføres til eierne av plattformen. Nets 3.7.5.1 er aktør i dette tilfellet og kommuniserer mot systemet på to punkter, både fra applikasjonsserver og mobilapplikasjon. Hvordan integrasjonen for betaling fungerer, forklares i detalj på Del 4.3.3.7.

4.3 Applikasjonsserveren

I denne seksjonen skal vi ta for oss hvordan vi satte opp applikasjonsserveren som mobilapplikasjonen kommuniserer med. Først beskriver vi hvorfor vi valgte å bytte applikasjonsserver halvveis ut i prosjektet. Deretter vil vi beskrive hvordan vi endte opp med å sette opp systemet og hva de forskjellige modulene gjør, samt deres oppgave.

4.3.1 Valg ved kodedesign

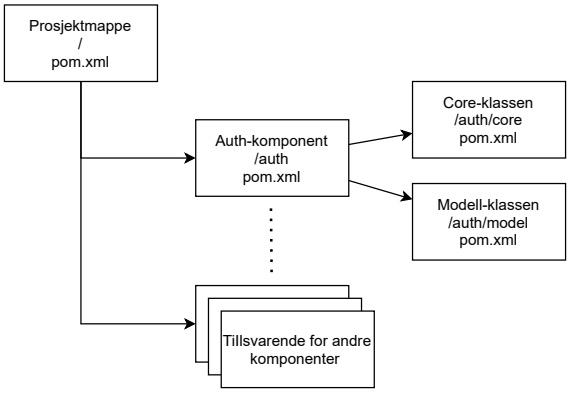
Her vil vi beskrive hvilke valg vi tok med tanke på kodedesign og struktur, hvordan vi satte opp filstrukturen, og hvilke prinsipper vi har valgt å følge med tanke på kode.

4.3.1.1 Filstruktur

Siden vi ønsket å splitte prosjektet opp i mikrotjenester, var det viktig at vi organiserte koden på et vis som var oversiktlig, lett å forstå, og lett å bygge. Vi ønsket også å ha en struktur som ga oss muligheten til å bygge i én operasjon i.e. ikke trenge spesifikke rekkefølger på bygg operasjoner eller kompliserte script. Det vi endte opp med var å bruke et flere lag med Apache sitt Maven 3.7.4.1 bygge system. Hele prosjektet bygges av en *multi-modulsprosjekt* POM-fil, også kjent som en POM pakket POM-fil. Denne byggefilen vil starte byggeprosessen for hver av de forskjellige tjenestene i den presiserte rekkefølgen.

Hvordan disse modulene bygges presiseres av hver modul's POM pakket POM-fil. Disse vil i all hovedsak først bygge tjenestens datamodeller. Disse er en definert som egne pakker etterfulgt av selve "Core" pakkene, som inneholder WAR-filene som blir distribuert til hver mikrotjenestene sin applikasjonsserver. Figur 4.3 viser hvordan POM filene er distribuert. Bakgrunnen for at vi har valgt å splitte det slik, stammer fra observasjonen om at vi i flere tilfeller trengte modell klassene fra andre moduler. E.g. *Group*-klassene, trengs i alle moduler for å definere tilgangsnivå. For å kunne importere disse klassene uten å dra med seg alle avhengighetene til Core WAR-filen, splittet vi disse. Hvordan selve Core og Model blir bygd, er definert i hver sin standard POM-fil. Core bygges til en distribuer bar WAR fil, og Model ender opp som en standard jar fil.

Hva dette oppsettet betyr, er at om man kjører byggekomandoen til maven på den øverste POM-en vil denne bygge hele prosjektet modul for modul i riktig rekkefølge med tanke på avhengigheter.



Figur 4.3: Maven-struktur for hver mikrotjeneste-komponent, her vist ved *Auth-komponenten.*

4.3.1.2 Null-sikkerhet i Java

For å tilnærme oss nullsikkerhet (2.5.1) på serveren som er skrevet i Java, valgte vi å bruke klassen `Optional`. Ved å bruke `Optional` i metoder som skal returnere noe, har vi alltid ett objekt som blir returnert. Dermed unngår vi at det returneres *null*, og vi slipper å havne i en situasjon hvor `NullPointerException` kan oppstå. `Optional`-objektet som returneres inneholder enten et objekt av den datatypen man er ute etter, eller så er det tomt. Kodeeksempel 4.1 viser et eksempel på hvordan vi har brukt `Optional` til å unngå å returnere *null*.

Kodeeksempel 4.1: Eksempel på bruk av Optional i vår kode

```
1    public Optional<Float> getUserRating(long userId) {
2        Query query = entityManager.createQuery(USER_RATING);
3        query.setParameter("rtd_id", userId);
4        try {
5            return Optional.ofNullable((Float) query.getSingleResult());
6        } catch (NoResultException e) {
7            return Optional.empty();
8        }
9    }
```

Å bruke `Optional` gjør det også enkelt å lage korte metoder som er nullsikker og oversiktlig. `Optional` sin *ofNullable()* metode lar oss med en linje kode lete i databasen etter et objekt, og så returnere et `Optional`-objekt. Hvis det ble funnet et passende objekt inneholder

`Optional`-objektet en beskrivelse av det, og hvis ikke er `Optional` tomt. På denne måten slipper vi å gjøre null-sjekker, fordi vi alltid får tilbake en forventet verdi. Et eksempel på dette er Kodeeksempel 4.2

Kodeeksempel 4.2: Eksempel på bruk av Optional i kort metode

```
1  public Optional<OfferListing> findOfferListingById(long listingId) {
2      return Optional.ofNullable(entityManager.find(OfferListing.class,
           listingId));
3  }
```

4.3.1.3 Validering av input-data

All data som kommer inn til serveren gjennom HTTP-forespørsler blir sjekket at de er av korrekt type og ikke er *null*. Dette gjøres gjennom å bruke Java-annoteringer fra JakartaEE-spesifikasjonen *Bean Validation* 3.7.1.7. Ved å annotere metode-parametere med f.eks `@NotNull` blir data fra HTTP-forespørsler sjekket før man går videre i metoden og bruker dem. På denne måten unngår vi at man kan la være å sende med krevet data og skape feil på serveren. Ved å annotere parametere på denne måten, sendes det automatisk et svar tilbake om at det ble sendt feil data til serveren. Å annotere metode-parameterne gjør det også slik at vi slipper å lage null-sjekker i metoden, og sende svar manuelt ved feil data. Dette fører til at koden blir mer lesbar og oversiktlig. Kodeeksempel 4.3 viser et eksempel på hvordan `@NotNull` brukes for å sjekke at serveren mottar en id når man prøver å hente en OfferListing.

Kodeeksempel 4.3: Eksempel på bruk av @NotNull for å sjekke input data

```
1  @GET
2  @Path("offer/{id}")
3  public Response getOfferListing(
4          @NotNull @PathParam("id") Long id) {
5      Optional<OfferListing> offerListing =
           listingService.findOfferListingById(id);
6      return offerListing.map(Response::ok)
7              .orElse(Response.status(Response.Status.NOT_FOUND)).build();
8  }
```

4.3.2 Monolitt-arkitektur

Vi startet med å lage serveren med en monolittisk-arkitektur 2.8.1. Vi gjorde dette for å få lavere kompleksitet på serveren og ha høyere ytelse mellom komponentene. På denne måten slipper en å gjøre HTTP-kall mellom komponentene og oppnår derfor høyere ytelse.

Etter hvert mens vi jobbet med oppgaven, valgte vi å gå over til en mikrotjeneste-arkitektur 2.8.2. Vi forteller mer om dette i neste avsnitt.

4.3.3 Mikrotjeneste-arkitektur

Det er totalt syv komponenter som virker sammen for å utgjøre serveren til handelsplattformen. Hver enkelt komponent kjøres i egen container 2.8.3.1. En slik mikrotjeneste-basert løsning ble valgt for å gjøre serveren enklere å vedlikeholde og mer skalerbar. Kommunikasjon mellom komponentene går over protokollen HTTP 2.9.3. Utad representerer serveren ett komplett REST API-grensesnitt 2.9.4 med endepunkter som brukes av mobilapplikasjonen og webgrensesnittet for administrasjon av plattformen. Nedenfor beskriver vi hvilke oppgaver vi hadde behov for at applikasjonsserveren skulle være i stand til å løse og hvordan vi designet containere rundt å løse disse oppgavene.

4.3.3.1 Auth

Vi innså tidlig at vi ville trenge en enhet som kunne håndtere autentisering av brukere og for gradering av tilgangs privileger. For å løse denne oppgaven lagde vi Auth containeren. Siden vi følger MicroProfile-spesifikasjonen 3.7.2 valgte vi å bruke JSON Web Token 3.7.4.8. For å kunne oppnå dette må Auth containeren være i stand til å tilby muligheten for brukere å logge inn. Deretter skal den kunne respondere med et JWT som inneholder informasjon om gruppene personen er med i. For at andre skal kunne validere tokenet må containeren også tilby den offentlige nøkkelen som kan brukes for å validere signaturen. For å kunne legge til nye brukere har containeren et eget endepunkt som kun andre container er validert til å bruke, hvordan er beskrevet i 4.3.3.8.

En annen fordel med å ha autentiseringsdataen i en egen database, er at data med høy ver-

di (brukernavn-passord) er mindre sårbar for SQL injeksjoner og andre angrep siden det er færre mulige angrepsvektorer inn til dataen.

4.3.3.2 User

Siden kravspesifikasjonen påkrevde at brukere skulle ha muligheten til å ha, og potensielt endre, sin bruker info anså vi dette som en den neste oppgaven vi trengte en enhet til å håndtere. Denne containeren har som oppgave å gi ut, og å kunne endre på brukerdata til både selgere 4.2.2 og kjøpere 4.2.1 i mobilapplikasjonen. User-containeren har også ansvar for å skape nye kjøpere og selgere. For å kunne gjøre dette kommuniserer den med Auth-containeren for å sjekke om brukernavnet er ledig. Hvis det er det, opprettes en ny bruker som lagres i databasen.

Bakgrunnen for å flytte disse oppgavene til en separat container, i stedet for å ha de i Auth-containeren, stammer hovedsakelig fra å prøve å unngå fremtidige bugs. I gruppens erfaring stammer ofte bugs fra høy kompleksitet eller dårlig gjennomførte "forbedringer"/nye funksjoner. Valget ble derfor tatt med å separere autentiseringen for seg selv og holde den så simpel som mulig. User-modulen som antakelig vil oppleve større grad av endring, vil da ha en mye lavere konsekvens for sikkerhetsbrudd. Selv med root tilgang i containeren vil man ikke ha annet en mail, navn o.l. Grunnen til dette er at passord og brukernavn er lagret annensteds.

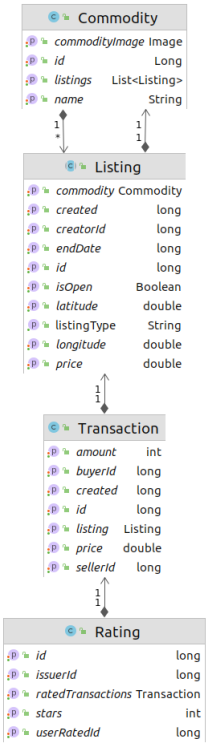
4.3.3.3 Store

Kjernefunksjonaliteten i vår applikasjon er muligheten for kjøp og salg av fisk og andre hav vesen. For å oppnå dette lagde vi Store-containeren.

Store-containeren har all funksjonalitet relatert til utførelse av en handel mellom kjøper og selger. Dette inkluderer håndtering av varekategorier, salgsannonser, kvitteringer og rangering av selgere. Varekategorier kan bare opprettes av en administrator gjennom et webgrensesnitt 4.3.3.6. En varekategori inneholder alltid et bilde av varen. Når Store mottar en forespørsel om å opprette en ny varekategori, splittes innholdet i forespørselen opp til to deler. En del som inneholder bildet og en del som inneholder informasjon om varekategori-

en. Bildet sendes til Media-komponenten 4.3.3.4 for å håndteres og lagres, og Store oppretter varekategorien og lagrer den i databasen med en referanse til bildet.

Vi valgte å la denne komponenten være så stor som den er på ytelsesgrunnlag. Kostnaden for å gjennomføre HTTP-kall er betydelig mye høyere, enn kostnaden for å gjennomføre funksjonskall. Datamodellene i Store klassen er avhengige av hverandre som vist i figur 4.4. Dette fører til at flere splittelser av klassen ville resultert i et stort antall HTTP-kall mellom containerene for å holde informasjonen synkronisert, og for å kunne gjennomføre de logiske operasjonene containeren må kunne gjøre.



Figur 4.4: Avhengihet forholdet mellom store modell klassene

4.3.3.4 Media

Siden nedlasting av bilder vil være den delen av applikasjonen vår som vil kreve den høyeste båndbredden, er det viktig at vi er i stand til å skalere oppgaven. Derfor lagde vi Media-containeren. Media-containeren håndterer lagring og henting av bilder. Komponenten kan

hente og sende tilbake både hele bilder, og generere miniatyrbilder etter behov. Klienter som ikke har behov for å hente ned bilder med full oppløsning, kan sende forespørsel om ønsket oppløsning, og Media-komponenten vil så svare med å skreddersy oppløsningen etter dette. Dette gjør at mindre datamengder overføres.

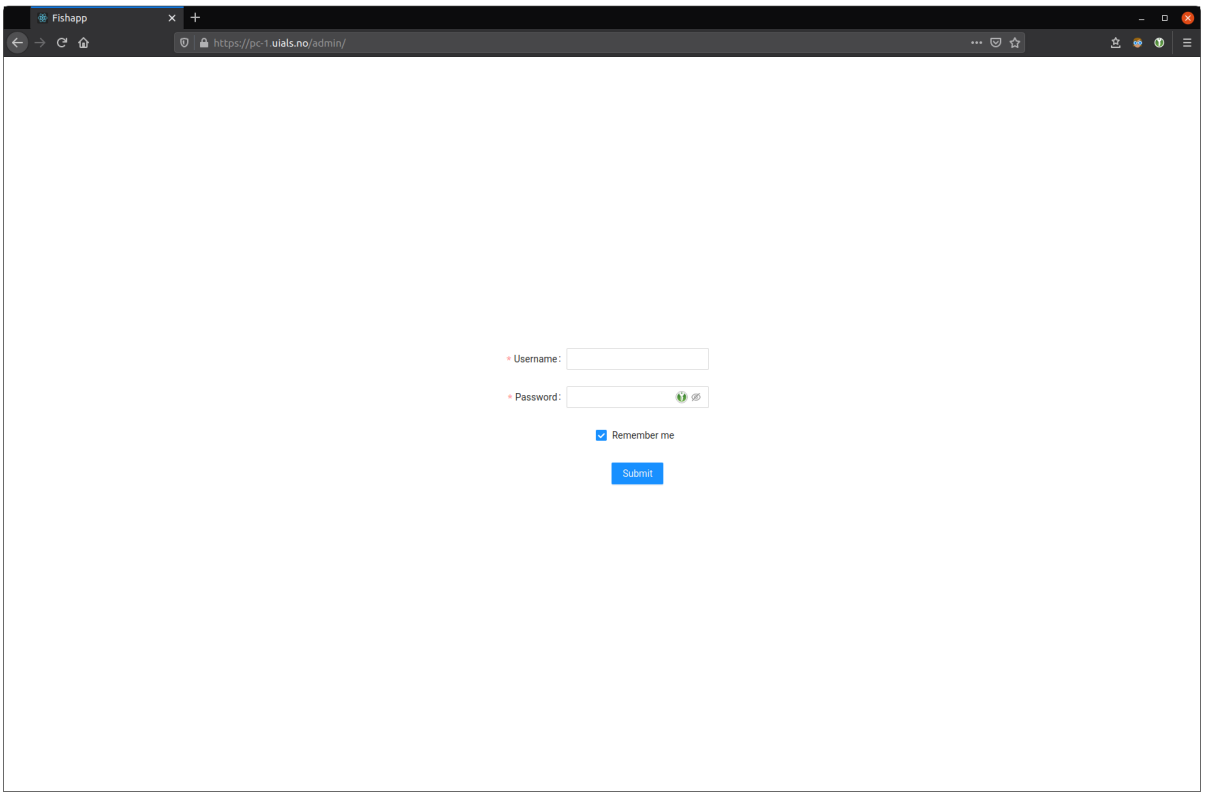
4.3.3.5 Chat

En forutsetning for å gjennomføre en handel, er god kommunikasjon mellom kjøper og selger. Ut ifra krav i oppgaveteksten og i samråd med oppdragsgiver ble det klart at det ville være avgjørende at tjenesten har funksjonalitet for å chatte mellom brukerne. Ved mange samtidige samtaler kan det bli betydelig last på serveren, selv om datamengden er liten.

Dette kan føre til tjenesten blir treg. For å ha mulighet for å skalere mere ressurser ved behov, ble bestemt å sette Chat i en egen komponent. Chat håndterer opprettelse av samtaler, og overføring av meldinger mellom kjøper og selger på plattformen.

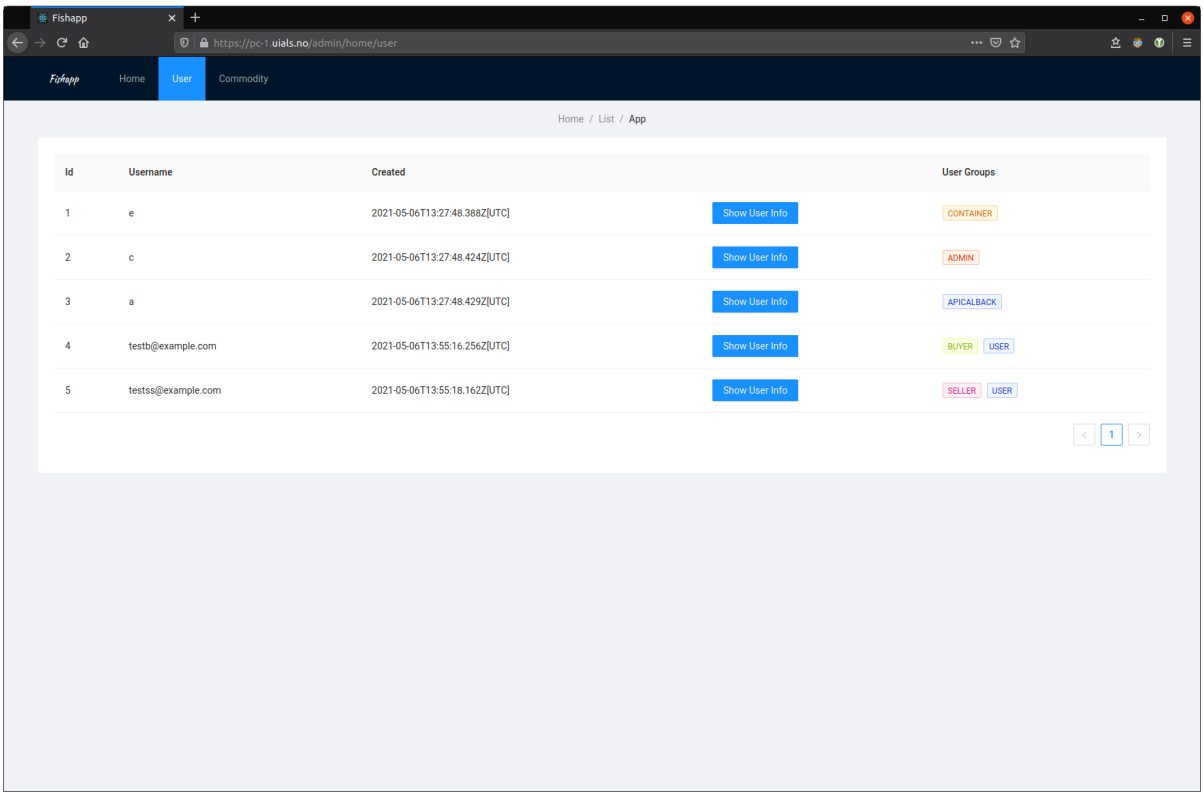
4.3.3.6 Frontend

For å kunne ha muligheten til å administrere brukere, få info om de forskjellige varetypene til salgs i systemet, og annen administrasjon trengte vi et eget webgrensesnitt for dette. Siden vi hadde allerede bestemt oss for å bruke mikrotjenester, valgte vi å ha webgrensesnittet driftet fra en egen container, nemlig frontend-containeren. Denne containeren er en simpel OpenLiberty-instans som drifter en singel-side webapplikasjon. Denne siden er laget i JavaScript-rammeverket React.js 3.9.1. Webgrensesnittet inneholder en enkel påloggingsside vist i figur 4.5.



Figur 4.5: Påloggingsside i webapplikasjon

Etter du har logget inn kommer du til startsiden for webapplikasjonen. Her har man flere forskjellige menyer for å navigere til de forskjellige administrering sidene. Webapplikasjonen er svært åpen for fremtidig modifikasjon, basert på hva oppdragsgiver finner ut er praktisk å ha tilgang til. Det er bare å legge til en ny fane og putte inn en meny. Et eksempel på en av disse admin sidene, er brukerinfo-siden hvor man får opp alle de forskjellige brukerne, hvilke roller de har i systemet og muligheten til å fjerne de. Denne siden kan sees i figur 4.6



Figur 4.6: Brukerinfo-side i webapplikasjon

Siden webapplikasjonen er laget i React, kan ikke denne bygges med Maven direkte. Å gjøre dette manuelt ville ha vert ugunstig, siden det betyr at å bygge prosjektet vil kreve flere operasjoner, noe vi anså som unødvendig komplekst. Vi valgte derfor å integrere byggings- og nedrivnings-prosessen inn i Maven. For å bygge webapplikasjonen underveis i kompileringen, brukte vi *Maven Exec Plugin* (Lacoste 2020), Denne kjører i gang scriptet vist i Kodeeksempel 4.4.

Kodeeksempel 4.4: frontend build scriptet

```
1 #!/bin/bash
2
3 export UID=$(id -u)
4 export GID=$(id -g)
5 docker build ./src/main/liberty \
6   -f ./src/main/frontend/build.Dockerfile \
7   -t fishapp-frontend_builder:latest \
8
9 docker run \
```

```
10   -t \
11   --rm \
12   --env UID=$UID \
13   --env GID=$GID \
14   --mount type=bind,source=$PWD/src/main/frontend,target=/node_app \
15   --name fishapp_frontend_builder fishapp-frontend_builder:latest
```

Dette scriptet vil bygge en container hvor Yarn er installert. Vi navigerer til containeren sin arbeidsmappe, i stedet for å kopiere filene inn ved bygging. Dette er så vi ikke trenger å kopiere *node modules* over til bygge-containeren men kan montere direkte fra filsystemet i stedet, som vil spare tid. Deretter vil denne containeren kjøres for å bygge React appen. For å unngå rettighetsproblemer på filkatalogene for *node modules* og build-filene, kjøres en rekursiv chown-kommando over til gruppen og brukeren gitt ved parameterne på linje 12 og 13 i Kodeeksempel 4.4. Docker-filen som bildet bygges fra er vist i Kodeeksempel 4.5

Kodeeksempel 4.5: frontend build docker filen

```
1 FROM node:15-buster
2 RUN apt update && apt install -y yarn
3 VOLUME /node_app
4 WORKDIR /node_app
5 CMD yarn install && \
6  yarn build && \
7  chown $UID:$GID -R /node_app/node_modules && \
8  chown $UID:$GID -R /node_app/build
```

4.3.3.7 Checkout

Den siste komponenten sin jobb er å håndtere betaling i applikasjonen. En av fordelene med at denne komponenten er separert, er at vi kunne skru av JPA 3.7.1.3 sin caching her uten å påvirke ytelsen til noen andre moduler. Selv om flere parallelle instanser kjører i Kubernetes, vil man alltid få den siste oppdaterte informasjonen på abonnement statusen til en bruker. Finansieringsmodellen for handelsplattformen endte opp med å være en abonnementsløsning, slik at selgere kjøper tilgang kunne legge ut sine salgsannonser i mobilappli-

kasjonen. For å få til dette, endte vi opp med å bruke Nets (*Den foretrukne betalingspartneren — Nets* 2021) som betalingsleverandør. Nets tilbyr en tjeneste de kaller EasyAPI (*DIBS Easy E-commerce - DIBS Technical Documentation* 2021) som er et API for å prosessere kortbetaling. Vi valgte å bruke denne løsningen, siden den var enkel å implementere for å sette opp gjentakende betalinger. Checkout-komponenten eksponerer et grensesnitt med tre funksjoner:

- Sjekke om en oppgitt bruker-id har et aktivt, godkjent abonnement.
- Be om et nytt abonnement for en bruker og få den genererte lenken til betalingssiden tilbake
- Kansellere et aktivt abonnement.

Måten abonnement fungerer i Nets EasyAPI er at man lager en betaling med en abonnement-id som man kan belaste flere ganger, i vårt tilfelle 1 gang i måneden, og få betaling. For å effektivt kunne få info fra Nets om hvilke faser i betalingsprosessen brukeren har passert, satte vi opp et sett med webhook'er2.9.8, hvor vi får info om den initiale betalingen var en suksess eller ikke.

Et problem vi sto ovenfor, var når det kommer til belastning av brukere. Siden det potensielt er flere instanser som kjører samtidig, vil dette medføre at belastningsprosessen vil kjøres flere ganger. Ved opprettelse av abonnementsinstanser hos Nets, setter vi minimum intervall mellom belastinger til å være 2 dager. På denne måten er det ingen sjanse for at brukere faktisk blir belastet 2 ganger. Likevel er det gunstig å være varsom, og sikre seg at slike feil ikke oppstår. Derfor har vi laget ett avstemmingssystem som hver av abonnementsinstansene blir med i, før belastningsprosessen starter. Avstemmingssystemet fungerer slik:

1. Generer en tilfeldig 32-bit heltall
2. Putt denne tilfeldige verdien inn i en stemmeseddel og lagre den til databasen (caching er slått av)
3. Vent i ett sekund, så alle er ferdig med stegene over.
4. Hent inn listen med alle sedlene, og sjekk om tallet i din seddel er:

- Større enn alle andre -> da har du ansvar for belastning av abonneringene.
- Nest størst -> da har du ansvar for å verifisere at den over har gjort sin jobb
- ingen av de 2 over -> slett seddelen og avslutt oppgaven

4.3.3.8 Utility

Utillity modulen består av to deler. Den første delen er: diverse utility-delen som inneholder forskjellige klasser, som en *multipart form handler* for å håndtere IMultipartBody. IMultipartBody er måten OpenLiberty bruker for å håndtere *HTTP-Multipart*-forespørsler. Forskjellige *exeption mappers* er også der.

Den andre modulen håndterer autentisering, ved hjelp av MicroProfile Rest Client. Siden vi valgte å bruke mikrotjenester fikk vi et behov for å kunne kommunisere med andre moduler i prosjektet. For selve klienten brukte vi MicroProfile Rest Client 3.7.2.1. Dette var blant annet fordi vi følger MicroProfile-spesifikasjonen 3.7.2 så modulen er allerede på applikasjons serveren. Selve Rest Client-modulen tar seg bare av å sette opp, gjøre kall, og rive ned klienten. Siden vi bruker JakartaEE security 3.7.1.8 for å definere brukerrettigheter gjennom MicroProfile JWT 3.7.2.3, trengte vi et system for å gi denne klienten et JWT med de korrekte brukergruppene for å sikre kommunikasjon mellom mikrotjenestene. Siden disse klientene kom til å bli brukt i nesten alle mikrotjenestene, må løsningen være en frittstående modul som kan legges til om nødvendig. MicroProfile Rest Client har funksjonalitet for å legge inn egendefinerte *HTTP-headere* til alle objektene fra en metodereferanse. Derfor endte vi opp med å implementere en textitinterface som de forskjellige MicroProfile Rest Client inteface-ne kan utvide. Vi kalte denne klassen `AuthBaseClientInterface` vist under i Kodeeksempel 4.6.

Kodeeksempel 4.6: AuthBaseClientInterface interfacen

```
1 public interface AuthBaseClientInterface {
2     default String getAuthToken() {
3         return RestClientAuthHandler.getInstance().getAuthTokenHeader();
4     }
5 }
```

For at `AuthBaseClientInterface` skal kunne tilby JWT til klientene som tar i bruk *interfacet*, trenger den en klasse som kan logge inn til Auth-komponenten 4.3.3.1 og være i stand til logge inn på nytt om *tokenet* nærmerer seg utløpsdato. Som man kan se på linje 3 i 4.6 brukte vi en Singelton 2.3.2 klasse `RestClientAuthHandler`. Denne er vist i Kodeeksempel 4.7 (kommentarer og logging er fjernet for kompakthet)

Kodeeksempel 4.7: RestClientAuthHandler klassen

```
1  @Singleton
2  @Startup
3  public class RestClientAuthHandler {
4
5      // singelton pattern
6      private static RestClientAuthHandler instance;
7      public static RestClientAuthHandler getInstance() {return instance;}
8
9      ... injection of fealds
10
11     private String tokenString;
12
13     private PublicKey jwtPubKey;
14     private boolean isKeyValid = false;
15
16     @PostConstruct
17     public void startup() {
18         instance = this;
19         tokenLoop();
20     }
21
22     public boolean isReady() {
23         return isKeyValid && tokenString != null;
24     }
25
26     public String getAuthTokenHeader() {
27         return tokenString;
```

```
28      }
29
30      @Asynchronous
31      private void tokenLoop() {
32          Instant refreshTime;
33          try {
34              if (! isKeyValid) {
35                  this.jwtPubKey = getTokenPubKey();
36                  this.jwtParser = buildJwtParser();
37              }
38              String authHeader = getLoginToken();
39              this.tokenString = authHeader;
40              refreshTime = this.getRefreshTime(authHeader);
41          } } catch (RestClientHttpException e) {
42              log.log(Level.WARNING,
43                      String.format("Conection http %s error geting inter container
                            login token. Retrying in 10s",
44                                  e.getResponse().getStatus()));
45              refreshTime = Instant.now().plus(10, ChronoUnit.SECONDS);
46          } catch (SignatureException e) {
47              log.log(Level.WARNING, "Error validating inter container login
                    token. refreshing pub key");
48              isKeyValid = false;
49              refreshTime = Instant.now().plus(1, ChronoUnit.SECONDS);
50          } catch (ConnectException e) {
51              log.log(Level.WARNING, "Error fetching key. retying in 5 sec");
52              refreshTime = Instant.now().plus(5, ChronoUnit.SECONDS);
53          }
54          Duration waitTime = Duration.between(Instant.now(), refreshTime);
55          scheduledExec.schedule(this::tokenLoop, waitTime.getSeconds(),
                TimeUnit.SECONDS);
56      }
57
58    ... private method impl
```

```
59
60 }
```

Som man kan se på linje 17 til 20 i Kodeeksempel 4.7 vil det første denne klassen gjør (etter å sette seg opp som Singleton), være å starte programloopen som henter *JSON Web Token*. Siden klassen har en `@Startup`-annotering (se linje 1), vil denne være første som kjøres når modulen starter. Selve programloopen for å hente *tokens* (linje 31 - 56) er en asynkron metode, som planlegger en ny kjøring av seg selv på slutten for hver iterasjon. Grunnen for at loopen er asynkron, er for å unngå at klassen senker ned oppstartstiden til resten av modulen. Prosessen gjennom loopen består av følgende steg: (Feilhåndtering i form av exception er markert med unummererte underpunkter)

1. Sjekk om offentlig RSA-nøkkel man har er OK, hvis ikke hent ny.
   - Om ikke Auth-komponenten har startet, vil `ConnectException` bli kastet ved forsøk på å hente nøkkelen. Vi venter litt og prøver på nytt senere
2. Logg inn på Auth-komponenten 4.3.3.1 og returner verdien av *HTTP-headeren* `Authorization` man får tilbake.
   - Om det en annen *HTTP-kode* 2.9.3.2 enn 200 returneres, vil en `RestClientHttpException` bli kastet. Siden dette mest sannsynligvis er midlertidig, logger mann det, og avventer i 10 sec før man prøver på nytt
3. Les verdien av utløpstiden til tokenet og planlegg ny kjøring av programloopen som henter *JWT-tokens* 20 min før utløpstid er nådd.
   - Om ikke JWT-signaturen stemmer overens med nøkkelen som er lagret, anta at nøkkelen er feil og sett den som *invalid*, start så på nytt etter 1 sekund.

I Kodeeksempel 4.8 er det vist eksempel på bruk av `AuthBaseClientInterface`.

Kodeeksempel 4.8: Eksempel på bruk av AuthBaseClientInterface

```
1 @RegisterRestClient(configKey = "imageClient")
2 @Path("/api/media/image/")
3 @ClientHeaderParam(name = "Authorization", value = "{getAuthToken}")
```

```
4 public interface ImageClient extends AutoCloseable, AuthBaseClientInterface {
5 ...
```

4.3.3.9 Øvrige komponenter

I tillegg til de tidligere nevnte modulene, kjøres også en container for å drifte DBMS-et for PostgreSQL-databasene. Vi har valgt å gjøre det på denne måten ved utvikling, fordi det gir oss fleksibilitet til å ta opp og ned databasen etter behov, og ikke minst at vi slipper å kjøre en dedikert databaseserver på en virtuell maskin eller rent fysisk. Ved en driftssituasjon vil det være mer aktuelt å kjøre SQL-databasene på dedikerte maskiner, utenom Kubernetes eller som en *managed database* som leies hos en skyleverandør. Da kan man mer fleksibelt konfigurere DBMS-et med den redundans og sikkerhet som er ønskelig, og ikke minst sørge for at det regelmessig tas sikkerhetskopier av databasene.

4.3.3.10 Kommunikasjon mellom komponentene

Kommunikasjon mellom komponentene i mikrotjenesten gjøres med HTTP 2.9.3. Hver komponent sine *Resource*-klasser tilgjengeliggjør REST API-grensesnitter, som kobles opp mot sine respektive Service-ressurser i Kubernetes. Hver av Service-ressursene benyttes videre av eksterne klienter, men også internt av andre mikrotjenester. For at mikrotjenestene skulle ha mulighet til å sende forespørsler til de andre, måtte det lages en REST API-klient. Denne har vi implementert basert på MP Client fra MicroProfile-spesifikasjonen 3.7.2 og er forklart mer om i Del 4.3.3.8.

4.3.4 Kjøremiljø (med Kubernetes)

I denne delen vil vi beskrive hvordan kjøresystemet i for applikasjonen i Kubernetes er satt sammen.

4.3.4.1 Automatisk skalering

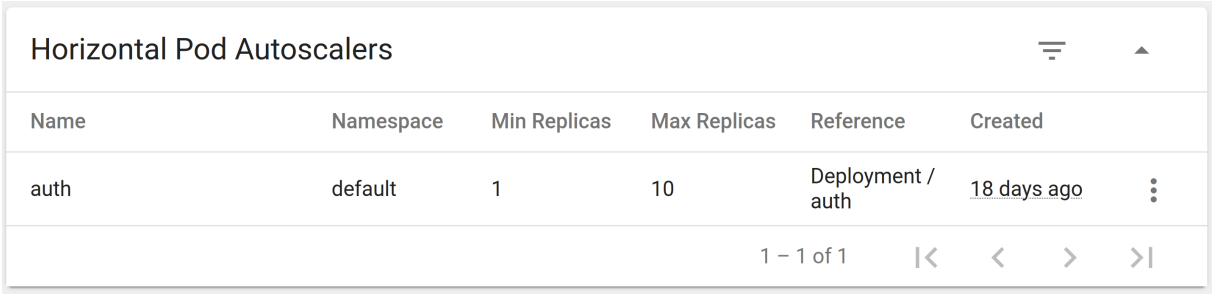
En av fordelene med å ha splittet applikasjonen opp i mindre deler, er at vi har muligheten til å skalere hver av delene individuelt om det skulle være behov for mer eller mindre ytelse. Siden vi valgte å bruke Kubernetes, var det enkelt å ta i bruk en slik skaleringsfunksjonalitet. Hvordan dette er spesifisert er vist i Kodeeksempel 4.9. Der kan man se på linje 10-12, at det er ønsket mellom 1 til 10 instanser, og at disse skal skaleres slik at enhver av Pod-ene 3.7.3.4 ligger på maksimalt 50 prosent av tildelt CPU-bruk.

Kodeeksempel 4.9: eksempel på hvordan HorizontalPodAutoscaler er definert her i auth modulen.

```
1  apiVersion: autoscaling/v1
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: auth
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: auth
10   minReplicas: 1
11   maxReplicas: 10
12   targetCPUUtilizationPercentage: 50
13 ---
14 apiVersion: apps/v1
15 kind: Deployment
16 metadata:
17   name: auth
18   labels:
19     app: fishapp
20 spec:
21   selector:
22     matchLabels:
23       app: fishapp
```

```
24      service: auth
```

I figur 4.7 kan man se hvordan skaleringstilstanden ser ut på Kubernetes-dashbordet for en mikrotjeneste. Her kan man se at det for øyeblikket er 1 instans startet opp, av den øvre grensen på 10 instanser.



Figur 4.7: Autoskaleringstilstanden for Auth-komponenten på Kubernetes-dashboardet.

Skaleringen vil vel og merke ikke være i drift for øyeblikket, fordi at den ikke slår inn hvis man ikke har definert CPU-grenser for containerne. Dette er et bevisst valg siden vi for øyeblikket kjører clusteret på en frittstående node 3.7.3.3 og vil derfor utnytte ressursene best ved å la alle containerne fritt dele alle tilgjengelige ressurser. I en produksjonssituasjon kan det være gunstig å sette slike grenser, så ikke en Pod bruker opp all kapasiteten til en node og påvirker andre Pod-er som kjøres. I et cluster med flere noder vil det også være en fordel at "deploymentene" 3.7.3.6 blir skalert, slik at lasten kan fordeles over flere noder.

4.3.4.2 Helse- og tilstandsrapportering

Siden Kubernetes er ansvarlig for å kontrollere, og starte og stoppe Pod-ene i clusteret må vi ha mekanismer for tilstandssjekker. De tilstandsjekkene vi har implementert er sjekker om Pod-er har startet opp, som vi kaller *ReadinessCheck*, og sjekerk på om en Pod-er i live, som vi kaller *LivenessCheck*. Disse sjekkene er implementert for alle moduler. *LivenessCheck*'en er lik for alle modulene, denne kan sees i Kodeeksempel 4.10. her bruker vi microprofile sin "liveliness"-modul som eksponerer endepunktet */health/live* og responderer med svaret vi returnerer på linje 10. Som man kan se på linje 13 vil sjekker returnere ok hvis den bruker under 90% av sitt allokerte minneområde. Men i hovedsak handler endepunktet om å sjekke om modulen *er i stand* til å svare, og minnesjekken er bare en bonus som vil slå ut om man

har noen lekkasjer.

Kodeeksempel 4.10: Eksempel på liveness sjekk for user modulen

```
@Liveness
@ApplicationScoped
public class UserLivenessCheck implements HealthCheck {
    MemoryMXBean memBean = ManagementFactory.getMemoryMXBean();
    long memUsed = memBean.getHeapMemoryUsage().getUsed();
    long memMax = memBean.getHeapMemoryUsage().getMax();

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("User Service Liveness Check")
                                  .withData("memory used", memUsed)
                                  .withData("memory max", memMax)
                                  .status(memUsed < memMax * 0.9).build();
    }
}
```

Readiness sjekkene vil til motsetning variere noe mellom de forskjellige modulene siden det er forskjell i hvordan vi vil beskrive tilstand klar i de forskjellige modulene. Variasjonen er i all hovedsak hvilke endepunkt det prøves et kall til og hvorvidt *RestClientAuthHandler* 4.3.3.8 brukes i modulen. De plassene hvor vår rest klient auth modulen brukes, inkluderer *ReadinessCheck*'en om rest auth handleren er klar, har logget inn, og har et valid token den kan gi til modulens rest klienter. Et eksempel på en slik sjekk er vist i Kodeeksempel 4.11.

Kodeeksempel 4.11: Eksempel på readyness sjekk for user modulen

```
@Readiness
@ApplicationScoped
public class UserReadinessCheck implements HealthCheck {

    private static final String readinessCheck = "User Service Readiness Check";

    @Inject
```

```
8     RestClientAuthHandler restClientAuthHandler;
9
10    public HealthCheckResponse call() {
11        if (isSystemServiceReachable()) {
12            return HealthCheckResponse.up(readinessCheck);
13        } else {
14            return HealthCheckResponse.down(readinessCheck);
15        }
16    }
17
18    private boolean isSystemServiceReachable() {
19        return requestOk() && restClientAuthHandler.isReady();
20    }
21
22    private boolean requestOk() {
23        try {
24            Client client = ClientBuilder.newClient();
25            client.target("http://localhost:9080/api/user/admin/ready").request().get();
26            return true;
27        } catch (Exception ex) {
28            return false;
29        }
30    }
31 }
```

På linje 22 kan man se at det prøves å sende et kall til containerens eget endepunkt. Om dette skjer uten problem og auth handleren sier den er klar (linje 19), anses containeren som å være i klar tilstand. Merk, kallet går til modulens localhost, som er intern trafikk i Pod-en. Dette er hvorfor kallet går over HTTP og uten TLS2.11.1.1.

Kodeeksempel 4.12: eksempel på hvodan *Liveness* og *Readiness* sjekk er definert i appen

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
```

```
4     ...
5 spec:
6     ...
7   template:
8     ...
9     spec:
10      ...
11      containers:
12          ...
13          readinessProbe:
14            httpGet:
15              path: /health/ready
16              port: 9080
17            initialDelaySeconds: 10
18            periodSeconds: 20
19            timeoutSeconds: 3
20            failureThreshold: 5
21          livenessProbe:
22            httpGet:
23              path: /health/live
24              port: 9080
25            initialDelaySeconds: 20
26            periodSeconds: 20
27            timeoutSeconds: 3
28            failureThreshold: 5
```

I Kodeeksempel 4.12 vises hvordan man definerer statussjekken i Kubernetes. Som man kan se på linje 16-19 definerer vi en initial forsinkelse til første test, periode, timout og antall tillatte feil. Om containeren bruker mer enn den definerte timout-tiden på å svare flere ganger enn hva som er tillat på rad, anses containeren som problematisk og vil bli erstattet med en ny instans.

4.3.4.3 Kryptering av inngående trafikk

Siden vi har innlogging i mobilapplikasjonen, vil det å sende logginn info i klartekst være et sikkerhetsproblem. Vi trengte derfor å implementere støtte for HTTP med TLS2.11.1.1. Enkleste måten å oppnå dette på, var å bruke cert-manager pakken 3.7.3.14. Denne pakken lar oss relativt enkelt implementere en Reverse Proxy 2.9.7, der trafikken krypteres med TLS2.11.1.1. Alt vi trenger å gjøre, er å definer en issuer av sertifikatet. Ett eksempel på dette kan sees i Kodeeksempel 4.13. Her definerer vi at vi ønsker å bruke http01 type utfordring (linje 39).

Kodeeksempel 4.13: Definering av sertifikat issuer

```
1  apiVersion: cert-manager.io/v1
2  kind: Issuer
3  metadata:
4    name: letsencrypt-prod
5    namespace: default
6  spec:
7    acme:
8      server: https://acme-v02.api.letsencrypt.org/directory
9      email: <<mail adress goes here>>
10     privateKeySecretRef:
11       name: acme-private-key-prod
12     solvers:
13       - http01:
14           ingress:
15             class: nginx
```

Kodeeksempel 4.14 viser hvordan vi satte opp vår ingress til å bruke issueren definert i eksempelet Kodeeksempel 4.13 på linje 6. På linje 7 setter vi opp denne ingressen til å alltid omdirigere vanlig HTTP trafikk til kryptert HTTP med TLS2.11.1.1. På linje 8 definerer vi at vi skal gjennomføre http01 utfordringen i denne ingressen og ikke generere en ny ingress kun for utfordringen.

Kodeeksempel 4.14: Definering av sefrikat til rev proxy

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    ...
  annotations:
    cert-manager.io/issuer: "letsencrypt-prod"
    ingress.kubernetes.io/ssl-redirect: "true"
    acme.cert-manager.io/http01-edit-in-place: "true"
spec:
  tls:
    - secretName: fishapp-tls-cert
      hosts:
        - pc-1.uials.no
  rules:
    - host: pc-1.uials.no
      http:
        paths:
          ...
```

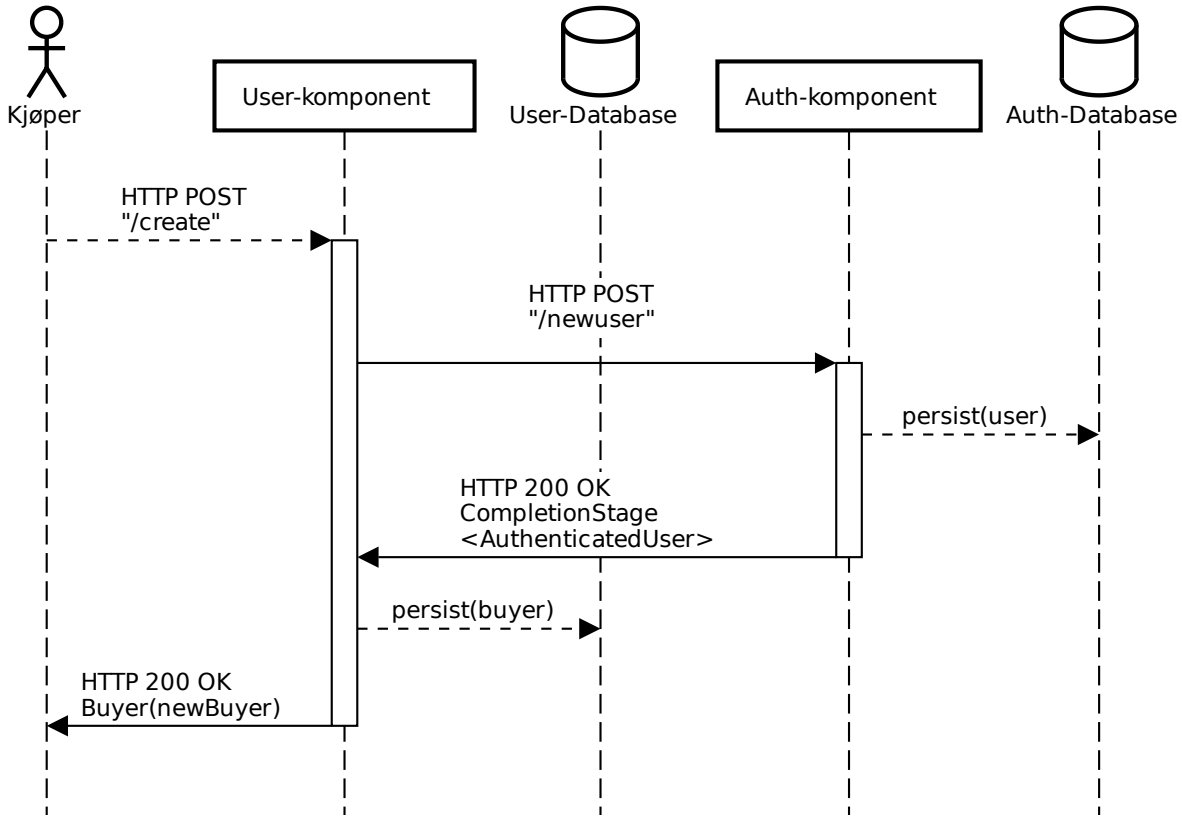
4.3.5 Serverens virkemåte

Her beskriver vi hvordan de forskjellige komponentene på serveren fungerer. Vi har også forklart hvordan sikkerhet er håndtert på serveren og hvordan data lagres. Til slutt har vi beskrevet hvordan vi har satt opp testing av serveren.

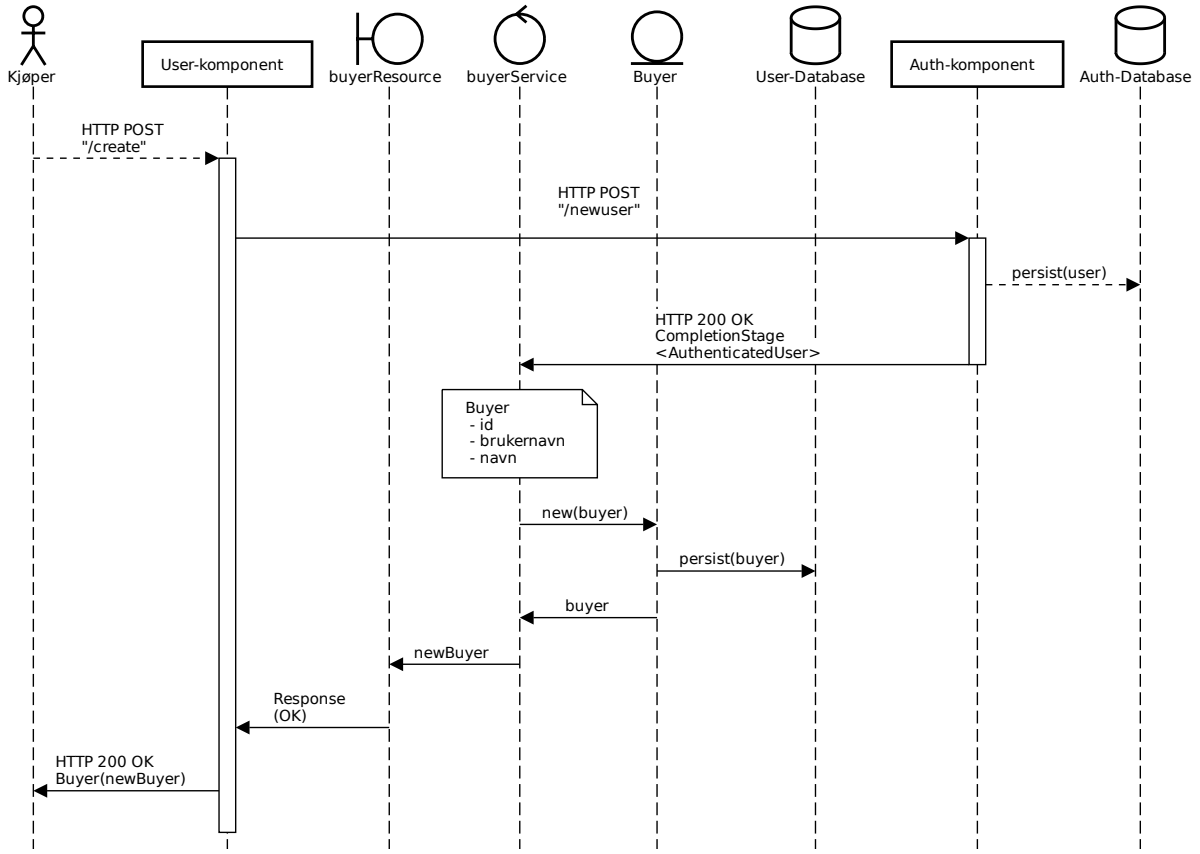
4.3.5.1 Brukerhåndtering

Håndtering av brukere for systemet foregår i *User-komponenten*, og gjøres ved hjelp av HTTP-kall 2.9.3 mot et REST API-grensesnitt 2.9.4. Grensenittet er definert etter brukergruppe, der selgere og kjøpere benytter sine respektive endepunkt. Ved forespørsel fra en kjøper om å registrere ny brukerkonto, vil dette gjøres i to trinn. Først vil forespørselen sendes fra brukerens mobilapplikasjonen til serveren, der den pekes til *User-komponenten*. Feltene for bru-

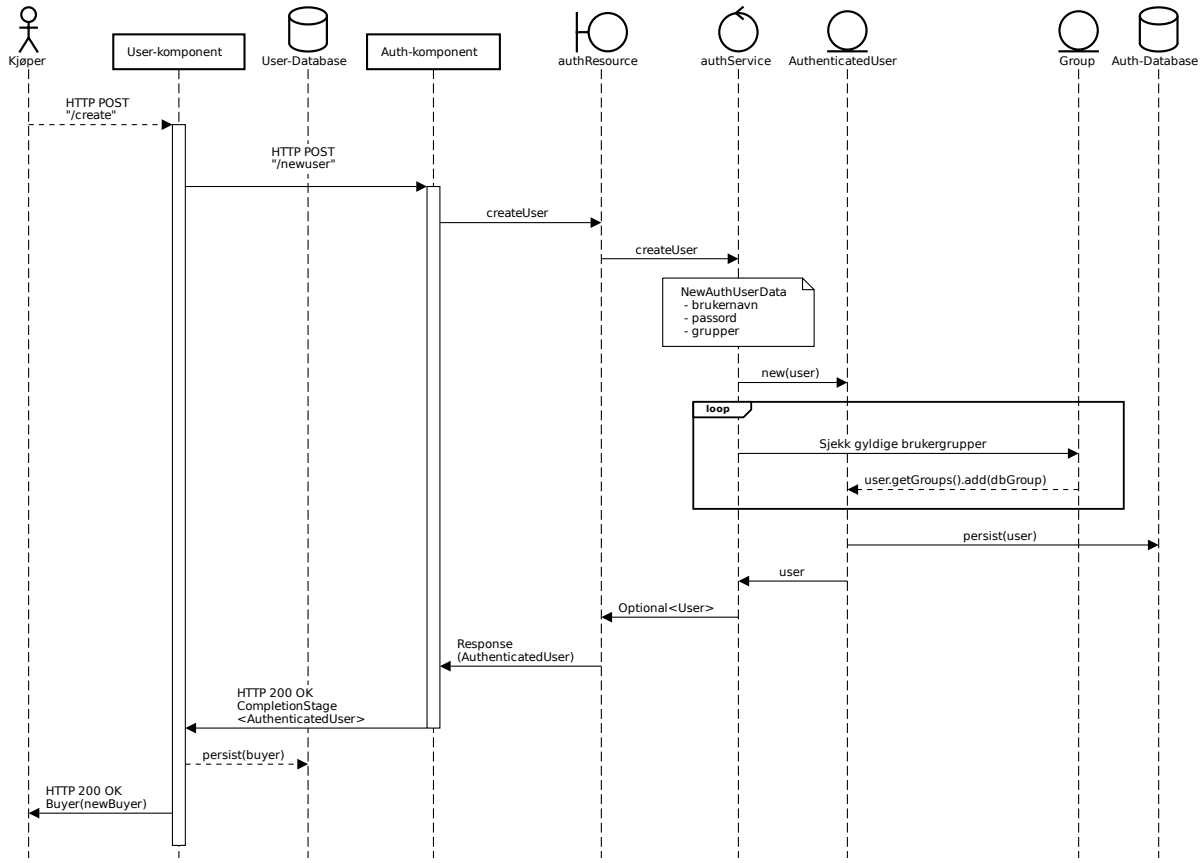
kernavn og passord sjekkes om de er korrekt skrevet og brukergruppene kjøperen skal ha blir satt. Som andre trinn sendes HTTP-kall med brukernavn, passord og gruppemedlemskap til *Auth-komponenten*. Brukernavnet kontrolleres så at det ikke er i bruk fra tidligere. Hvis brukernavnet er ledig, blir kontoen opprettet og lagret til *Auth-databasen*. *Auth-komponenten* sender svar tilbake til *User-komponenten* om at brukeren ble opprettet. Tilslutt lagres brukeren til *User-databasen*, og det sendes respons til mobilapplikasjonen om at alt er i orden og forespørselen ble utført. Figur 4.8 viser dette på et overordnet nivå, Figur 4.9, 4.10 går mer i detalj hvordan dette foregår innad i hver komponent.



Figur 4.8: Sekvens over brukerregistering mellom *User-komponent og Auth-komponent.*



Figur 4.9: Sekvens over brukerregistering innad i *User-komponenten*.

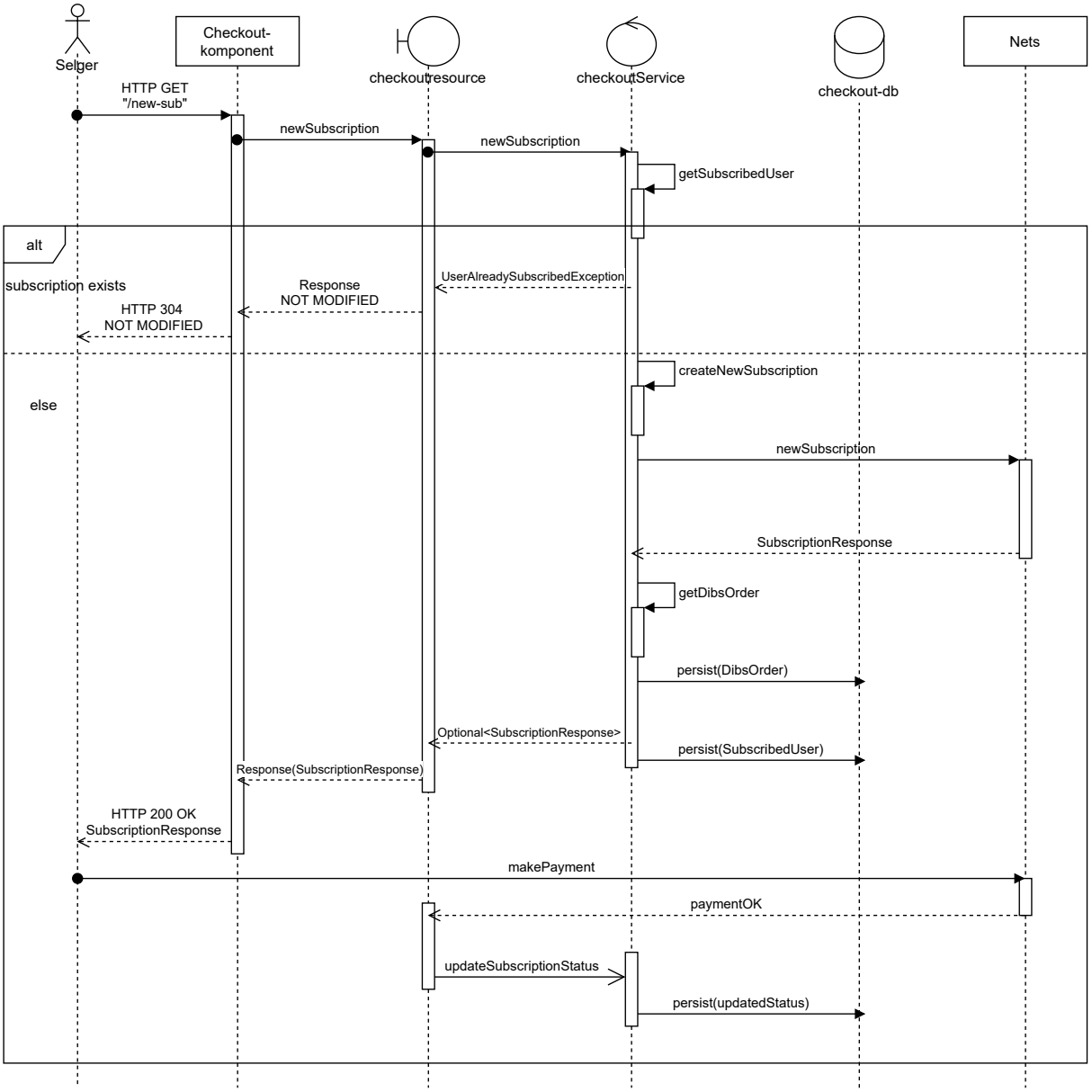


Figur 4.10: Sekvens over brukerregistering innad i *Auth-komponenten.*

4.3.5.2 Abonnementsløsning

For å kunne ha inntjening fra applikasjonen valgte vi å gå for en abonnementsløsning. For at selgere skal få muligheten til å legge ut salg må de ha et aktivt abonnement. Med et abonnement trekkes det ett månedlig beløp for tilgang til tjenesten. Figur 4.11 viser hvordan en selger starter et abonnement. Selger sender en forespørsel til **checkout**-komponenten om å opprette et abonnement. Dette mottas av *checkoutresource*, som sender det videre inn til *chekoutService*. Her hentes det det en *SubscribedUser* som er knyttet til selgeren som lagde forespørselen, og det sjekkes om den allerede har et abonnement. Om den har det, så sendes det et *NOT MODIFIED* svar til selgeren med informasjon om at de allerede har et abonnement. Hvis de ikke har et abonnement, opprettes det og det sendes en forespørsel til Nets om å opprette ett abonnement. Fra svaret som Nets sender tilbake, lages det en *SubscriptionResponse* og en *DibsOrder*. *DibsOrder* og *SubscribedUser* skrives til databasen, mens *Subscriptionresponse* sendes til applikasjonen til selgeren. Når mobilapplikasjonen til selgeren mottar *SubscriptionReponse*, åpnes det *WebView* med en URL som går rett til Nets. I

dette *WebView*-et legger selgeren til betalingsinformasjon, som de sender rett til Nets. Om betalingen er OK, sender Nets et svar til serveren om det og statusen til abonnementet blir oppdatert.



Figur 4.11: Sekvens over oppretting av nytt abonnement.

4.3.5.3 Samtaler mellom brukere i Chat

Samtaler som blir opprettet i mobilapplikasjonen knyttes mot brukerne i to forskjellige relasjoner. Brukeren som tar initiativ til samtalen, blir direkte definert som eier. Det blir også opprettet en knytning mot salgs- eller kjøpsannonsen (*listingen*) som samtalen ble opprettet fra. Samtalen vil da bli tilgjengelig for både brukeren som tok initiativ og eier av den tilknyt-

tede annonsen. På denne måten vil en samtale alltid ha to parter, og en enkelt kan finne ut hvilken annonse det dreier seg om.

Hver samtale inneholder en liste over alle meldinger som er sendt. Når en bruker oppretter en melding i mobilapplikasjonen, vil Chat-komponenten lagre denne og knytte denne til i samtalen. Alle meldinger som opprettes, blir lagret til databasen med en unik identifikator (ID). Det er dette tallet som knytter meldingen opp mot en samtale.

Mobilapplikasjonen benytter disse sekvenstallene til å holde styr på hvor mange meldinger den har fra før og hvor mange den trenger å laste ned. Ved spørringer mot server, vil mobilapplikasjonen sende sekvenstallet på siste mottatte melding. Chat-komponenten på server vil motta dette og kalkulere ut mengden manglede meldinger.

Antall manglede meldinger finnes ved regne differansen mellom sekvenstallet for siste melding på server og sekvenstallet som mobilapplikasjonen sendte. For hver forespørsel vil serveren regne ut korrekte sekvenstall, slik at mobilapplikasjonen alltid vil ha rette tall. Dette gjøres via en *stream-funksjon* som samler alle eksisterende meldinger, og så skriver om mengden til å bruke sekvenstall istedenfor.

4.3.6 Sikkerhet

For å sørge for tilstrekkelig sikkerhet, har vi fokusert på en del tiltak for å sikre serveren mot vanlige sårbarheter og feil. Nedenfor beskriver vi de viktigste tiltakene som er utført, i tillegg til sikkerhet som allerede er innebygget ved at vi har benyttet JakartaEE-rammeverket. Viktige punkter som er verdt å opplyse er at serveren benytter kryptert kommunikasjon med HTTP/TLS, og alle passord som lagres både saltes og hashes.

4.3.6.1 SQL-injeksjon

SQL-injeksjon 2.11.3 er en metode for å angripe applikasjonsserverer på Internett. Det er derfor sannsynlig at tjenesten og applikasjonsserveren under drift, kan bli rammet med et SQL-injeksjonsangrep. For å forhindre at dette blir utnyttet, har en benyttet JPA-rammeverket EclipseLink 3.7.4.6 til å håndtere forespørsler mot databasen. I utgangspunktet er ikke JPA

sikkert uten videre, men EclipseLink garanterer i sin dokumentasjon, at standard-innstillingene for Eclipslink er trygge mot SQL-injeksjonsangrep (Eclipse Foundation 2019a). Vi har derfor benyttet standard-innstillinger og *best practises* ved implementasjon, slik at applikasjonsserveren i mindre grad er sårbar for SQL-injeksjoner. Kodeeksempel 4.15 et eksempel på hvordan spørringer mot SQL er implementert.

Kodeeksempel 4.15: Utsnitt som viser oppbygging av spørring i JPA *(fra RatingService.java)*

```
1   public Optional<Rating> getRating(long transId, long issuId, long ratedId) {
2   ...
3     var query = entityManager.createQuery(GET_TRANSACT_RATING_QUERY,
        Rating.class);
4
5       query.setParameter("isu_id", issuId);
6       query.setParameter("rtd_id", ratedId);
7       query.setParameter("t_id", transId);
8       try {
9           return Optional.of(query.getSingleResult());
10      } catch (NoResultException ignore) {
11          return Optional.empty();
12      }
13  ...
14  }
```

4.3.6.2 Avlytting

All trafikk mellom mobilapplikasjon installert på brukerne sin mobiltelefon og applikasjonsserver vil gå over Internett. Dette er også tilfelle ved administrasjonsgrensesnittet, fordi det kjøres i brukernes nettleser. For sikre systemet mot avlytting, slik at ikke data kommer på avveie - krypteres all trafikk mellom brukerne og applikasjonsserver med protokollen TLS 2.9.3. Dette beskrives mer i detalj på kapittel 2.

4.3.7 Datalagring

Hver av mikrotjenestene benytter en PostgresSQL-driver for å koble til sin spesifikke, uavhengige database. Innloggingsdetaljer for databasen blir satt ved hjelp av OpenLiberty sin konfigurasjonsfil, men kan også konfigureres ved å sette miljøvariabler. Ved hjelp av Kubernetes kan også disse overstyres etter behov. Data innad i hver av mikrotjenestene lagres til SQL-basen med ORM-rammeverket EclipseLink, som vi har tatt i bruk.

Et par av mikrotjenestene som auth og media, har i tillegg behov for å lagre data direkte mot et filsystem. Fordi vi benytter containere til å kjøre mikrotjenestene, vil filer som er lagret inni containerne også gå tapt når en container fjernes eller opprettes på nytt. For at viktige filer ikke skal gå tapt, monteres deler av filsystemet innad i containeren som egne volumer, håndtert av systemet som kjører containeren.

Da vi benytter Kubernetes ble dette løst ved å lage PersistentVolumeClams og PersistentVolumes for containerne som trengte dette. På denne måten kan auth og media, både slettes og skaleres - uten at dette fører til datatap. Media benytter volumer for å lagre bilder og Auth holder på nøkkelparet som benyttes ved autentisering i mikrotjenesten. Sistnevnte er viktig når auth skal skaleres, fordi alle auth containerne må benytte det samme nøkkelparet for at de skal fungere som en enhet.

I utviklingssammenheng har vi benyttet PersistentVolumes som lagrer data til det lokale filsystemet på maskinen som kjører Kubernetes. Det har vært praktisk å lagre volumene på lokal maskin, siden de enkelt kunne slettes eller endres. Fordi vi har benyttet Kubernetes, gjør dette at andre volumtyper kan enkelt velges dersom det er ønskelig. Dette gjøres ved å endre volumet i definisjonsfilen til et Deployment. I en produksjonssituasjon der for eksempel mikrotjenesten kjøres fra en skyleverandør, kan en da benytte blokklagring fra den respektive leverandøren.

4.3.8 Testing

Testing av Jakarta EE applikasjoner har flere utfordringer sammenliknet med vanlig Java testing. Mye av disse problemene stammer fra at koden vi skriver bare er en liten del av det større Jakarta EE server systemet som kjører når man starter applikasjonen. Av den grunn,

for å kunne unit-teste 2.5.2.1 applikasjonen uten å starte opp en full applikasjonsserver måtte vi emulere punktene hvor koden vi tester samhandler med applikasjonsserveren. Måten vi valgte å løse dette problemet på var å bruke Mockito 3.11.1.2 og weld-junit 3.11.1.3 sammen med junit5 3.11.1.1.

Et eksempel på denne mocking'en kan sees i Kodeeksempel 4.16. Dette er en del av en større metode som kjøres før hver test. Fordi databasen ikke kjøres opp under unit-testing lages det her en mock av *EntityManager* som brukes for å kommunisere med databasen. Deretter settes mock'en opp slik at når testene kaller på metoder i *AuthenticationService* klassen som bruker *EntityManager* returneres det predefinerte objekter istedenfor at metoden kjører som den skal. For eksempel på linjene 10-13 defineres det at når metoden *createQuery* med spesifikke parametere kalles på mock objektet returneres det et predefinert brukerobjekt. Til slutt i kodeeksemplet blir mock'en "injected" inn i *AuthenticationService* klassen som skal testes.

Kodeeksempel 4.16: Eksempel på en mock av EntityManager

```
1  //  -- directly injected in class -- //
2      private EntityManager activeEntityManagerMock;
3
4  @BeforeEach
5  void buildMocks() {
6      ...
7      EntityManager entityManagerMock = Mockito.mock(EntityManager.class);
8
9          Mockito.when(entityManagerMock
10                             .createQuery(
11                                 AuthenticationService.GET_USER_BY_PRINCIPAL_QUERY,
                                       AuthenticatedUser.class))
12                 .thenReturn(userByPrincipalQ);
13
14         Mockito.when(entityManagerMock.find(Mockito.eq(AuthenticatedUser.class),
               Mockito.eq(USER_ID)))
15                 .thenReturn(defaultAuthUser);
16     ...
17     activeEntityManagerMock = entityManagerMock;
```

```
18          weld.select(AuthenticationService.class).get()
19          .setEntityManager(activeEntityManagerMock);
20      ...
21  }
```

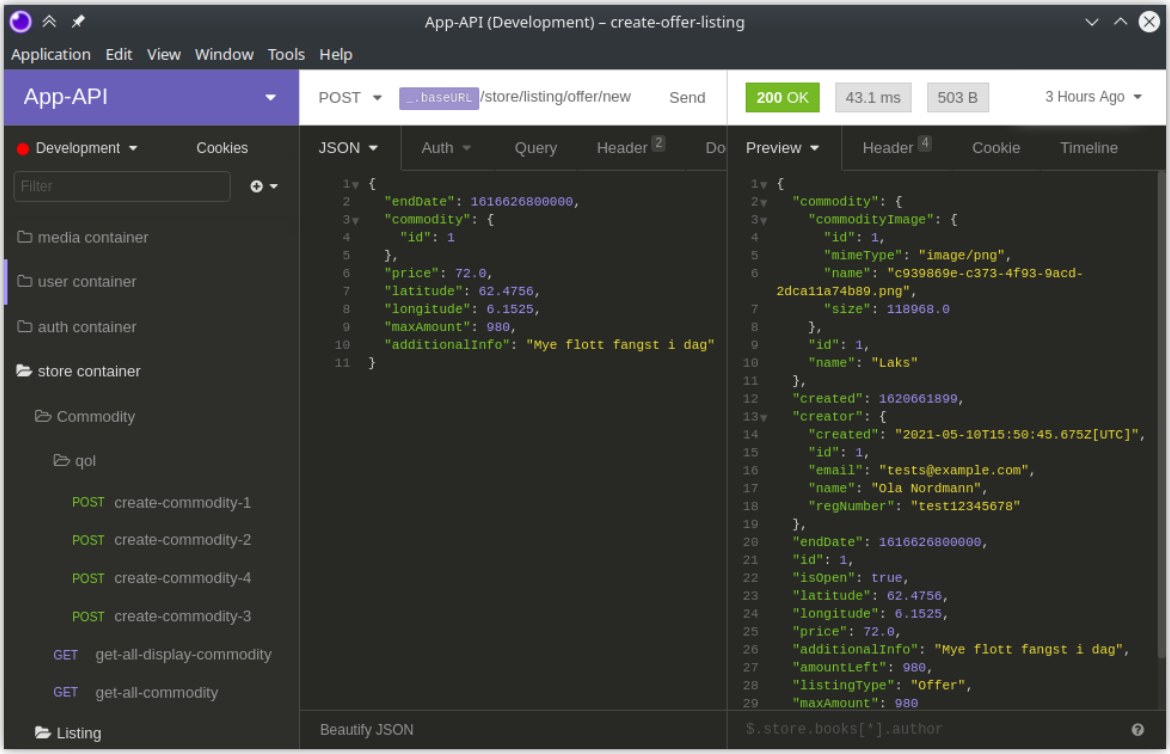
I Kodeeksempel 4.17 vises en test av *AuthenticationService*. Metoden *getUserFromId* bruker mock objektet fra forrige kodeeksempel og kan derfor returnere et forutsigbart resultat uten å faktisk spørre databasen etter en spesifikk bruker. Denne testet tester at metoden fungerer som forventet både hvis den finner en bruker med spesifisert id og hvis den ikke finner noen.

Kodeeksempel 4.17: Eksempel på en unit test

```
1  @Test
2      void getUserFromId() {
3          Optional<AuthenticatedUser> positiveTest =
               weld.select(AuthenticationService.class).get()
4                                                   .getUserFromId(USER_ID);
5          assertTrue(positiveTest.isPresent());
6          assertEquals(positiveTest.get().getId(), USER_ID);
7
8          Optional<AuthenticatedUser> negativeTest =
               weld.select(AuthenticationService.class).get().getUserFromId(1);
9
10         assertTrue(negativeTest.isEmpty());
11     }
```

4.3.8.1 Testing med Insomnia

Underveis i utviklingen av API-er vil det være behov for å teste dem fortløpende for å kunne verifisere at de faktisk virker. Av den grad valgte vi å ta i bruk GUI-verktøyet Insomnia 3.13.3. Insomnia kunne kjøres på lokalt på forskjellige operativsystemer og støttet lagring i klartekst til git-vennlige filformater som YAML. Dette gjorde det svært enkelt for oss å utvikle REST-API-et sammen, og alle kunne til enhver tid se hva som var gjeldende endepunkter, og ikke minst teste at det fungerte.



Figur 4.12: Testing av POST-forespørsel i Insomnia

4.4 Mobilapplikasjon

Mobilapplikasjonen er selve utgangspunktet til prosjektet, og i dette kapittelet gjennomgås det hvordan mobilapplikasjonen er oppbygget. Sentral funksjonalitet, brukergrensesnitt, kommunikasjon og sikkerhet fortelles det mer om.

4.4.1 Brukergrensesnitt

I dette avsnittet vil vi skrive litt om hvordan vi utviklet brukergrensesnittet til mobilapplikasjonen. Dette gjorde vi etter utgangspunkt fra oppgaveteksten og ved hjelp av tilbakemeldinger fra oppdragsgiver underveis i prosjektet.

4.4.1.1 Designprosessen

Prosessen for å lage brukergrensesnittet tok utgangspunkt i oppgaveteksten som ble gitt av oppdragsgiver ved oppstarten av prosjektet. Oppgaveteksten definerte gode føringer på hvilken funksjonalitet som var ønskelig å ha med. Ut i fra dette konkretiserte prosjektgruppen en kravspesifikasjon i Jira 3.3.6 over funksjonelle krav, inkludert hva en anså som nødvendige skjermbilder for brukergrensesnittet. Underveis i dette arbeidet ble det satt prioritering for det aktuelle kravet, definert som "*må ha* / *fint å ha*". For hvert av kravene opprettet vi del-krav for hvordan de ulike sidene skulle implementeres.

Dette var i midten av januar, og på denne tiden var ikke oppdragsgiver tilgjengelig for et møte grunnet arbeid offshore. De viktigste spørsmålene og synspunktene ble derfor først kommunisert over e-post. Etter tilbakemelding, ble kravspesifikasjonen justert og gikk prosessen videre med å opprette del-krav for hvordan de ulike sidene skulle implementeres.

Det ble opprettet egne del-krav for utarbeidelse av hvert skjermbilde i brukergrensesnittet. Arbeidsmengden her ble fordelt omtrent likt på alle i utviklingsteamet. Selve designprosessen var to-delt, og omhandlet både skissering av design og implementasjon av koden i Flutter 3.8.1.2. Først skisserte en designforslag som *wireframes* i Adobe XD 3.12.1. Deretter ble forslaget vektet og diskutert innad i gruppen, og eventuelle justeringer gjort til alle var fornøyde med designet.

Når vi hadde fått et godt utgangspunkt for videre arbeid, ble alle *wireframes* sendt på e-post til oppdragsgiver for vurdering. Etter tilbakemeldinger og justeringer etter oppdragsgivers ønske, startet selve arbeidet med å implementere programkoden som har ansvar tegne opp brukergrensesnittet i mobilapplikasjonen. I dette arbeidet ble det lagt stor vekt på å implementere gjenbrukbare komponenter. Dette gjorde prosessen vanskeligere og mer tidkrevende. En stor fordel derimot er at tid brukt her, kan man spare inn ved at fremtidig vedlikehold kan gjøres enkelt med færre endringer.

4.4.1.2 Designvalg

For at brukergrensesnittet skulle føles naturlig å bruke på tvers av to plattformer, bestemte vi oss tidlig for å lage vårt eget design. På denne måten favoriserer ikke designet plattformene

det kjøres på, og det vil oppleves som nøytralt av brukeren. Underveis i utformingen tok vi utgangspunkt i litteratur og velkjente prinsipper 2.6 for hva som utgjør et godt brukergrensesnitt. Fordi applikasjonen også kjøres på iOS-plattformen der det ikke finnes tilbakeknapp, måtte vi også ta hensyn til å implementere denne på alle skjermbilder i applikasjonen.

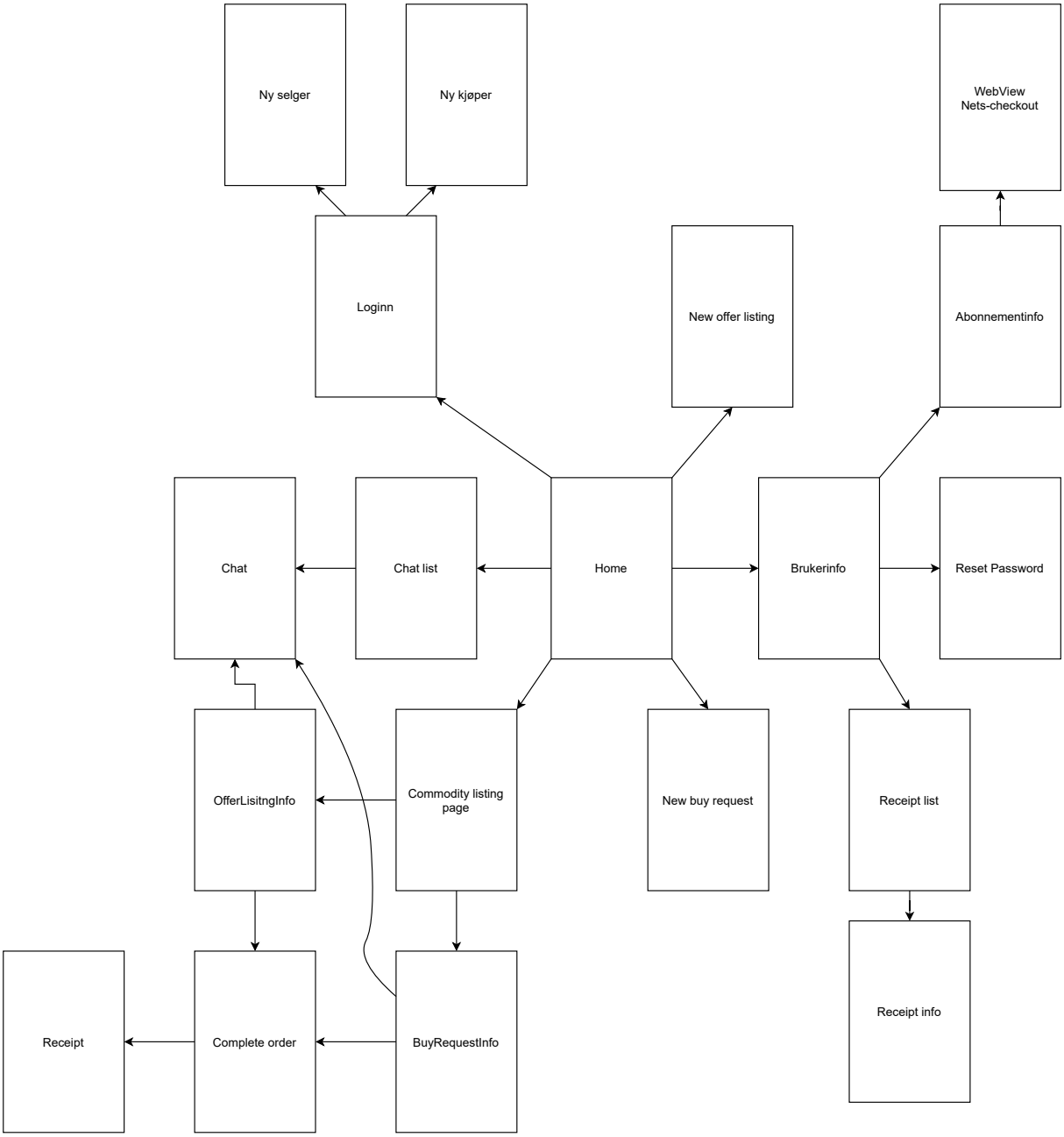
4.4.1.3 Feilhåndtering

Et viktig aspekt ved applikasjonen er at den skal være enkel å bruke. Det er viktig at alle feil som kan forekomme blir håndtert av brukergrensesnittet på en fattet måte, uten unødvendig informasjon. 2.6.1. Fordi applikasjonen avhenger av kommunikasjon med server over nettverk og er tenkt brukt med mobilnett, er det en viss risiko for brudd på tilkoblingen mellom applikasjon hos bruker og server.

En slik situasjon kan gi en ukjent tilstand på data i applikasjonen. Derfor har et viktig aspekt under utviklingen vært å ta hensyn til at slike brudd kommer til å skje og at brukergrensesnittet kan håndtere dette. Vi har derfor vektlagt å vise tydelige og kort formulerte feilmeldinger hvis brudd skulle oppstå. I likhet med resten av applikasjonen er meldingene lokaliserte 2.6.2 og skrevet på brukeren sitt språk, slik at de kan tolkes på en enkel måte.

4.4.1.4 Navigasjon

Oversiktlig navigasjon er sentralt for at applikasjonen skal være enkel å ta i bruk. Derfor har en i stor grad lagt vekt på å ha et synlig navigasjonsfelt med høy kontrast og få elementer. Navigasjonen i appen foregår i en hierarkisk tre-struktur, ved hjelp av en *router*-komponent. Figur 4.13 viser navigasjonsgrafen til mobilapplikasjonen.



Figur 4.13: Navigasjonsdiagram for mobilapplikasjon.

Ved beslutning om navigasjon, enten fra valg gjort av bruker eller logikk i applikasjonen - blir *Router*-komponenten koblet inn, og denne gir beskjed om at Flutter 3.8.1.2 skal tegne et nytt skjermbilde. Skulle det være at brukeren navigerer til en side som krever innlogging, men brukeren faktisk ikke er logget inn - vil *router*-komponenten sende brukeren til påloggingsskjermbildet. Her kan du logge inn eller lage bruker.

Navigasjonsfeltet er synlig på alle av applikasjonens hoved-sider, og viser aktivt med farge på hvilken av hoved-sidene brukeren oppholder seg på. Dette gjør det enkelt å vite hvilken

del av applikasjonen en befinner seg i, og stemmer godt i tråd med Don Normanns brukbarhetsprinsipper 2.6.1 om visuell tilbakemelding.

4.4.2 Funksjonalitet

Mobilapplikasjonen sin virkemåte er i stor grad basert på prosjektet sin kravspesifikasjon. Denne ble utarbeidet av informasjon gitt av oppgaveteksten og diskusjon mot oppdragsgiver. Den viktigste funksjonaliteten er forklart i avsnittene nedenfor.

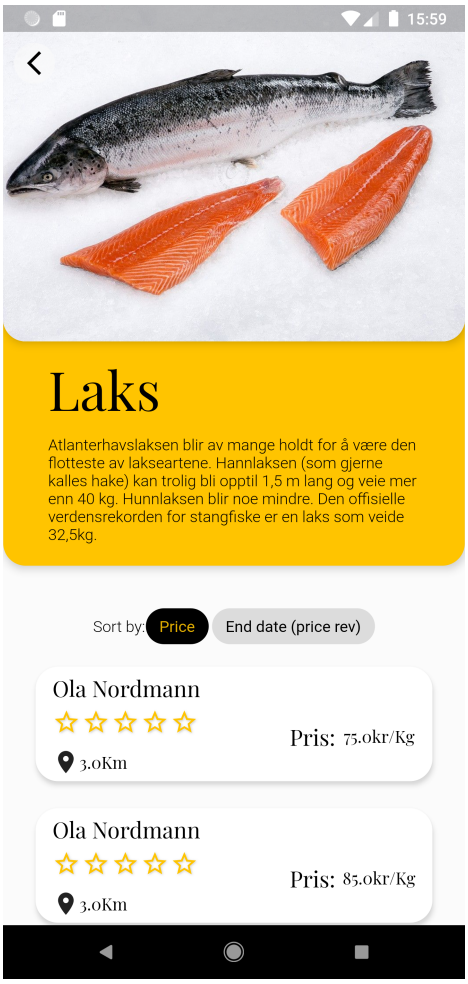
4.4.2.1 Navigasjon

Navigasjon i applikasjonen starter på hovedsiden. På hoved siden vises alle varekategorier som har en salgsannonse knyttet til seg. Siden har mulighet for å søke gjennom alle varekategoriene med et søkefelt. Det er også mulig å bruke filtre for å bare vide *fisk* eller *skalldyr*. Nederst på siden er det en navigasjonsbar som man bruker for å navigere seg mellom hovedsiden, chat, en brukerprofil, og å opprette nye annonser. Figur 4.14 viser et skjermbilde av hovedsiden i applikasjonen. Hvis man trykker på en av varekategoriene åpnes en liste med alle annonsene som tilhører den kategorien.

Figur 4.14: Hovedsiden i applikasjonen

4.4.2.2 Se gjennom annonser

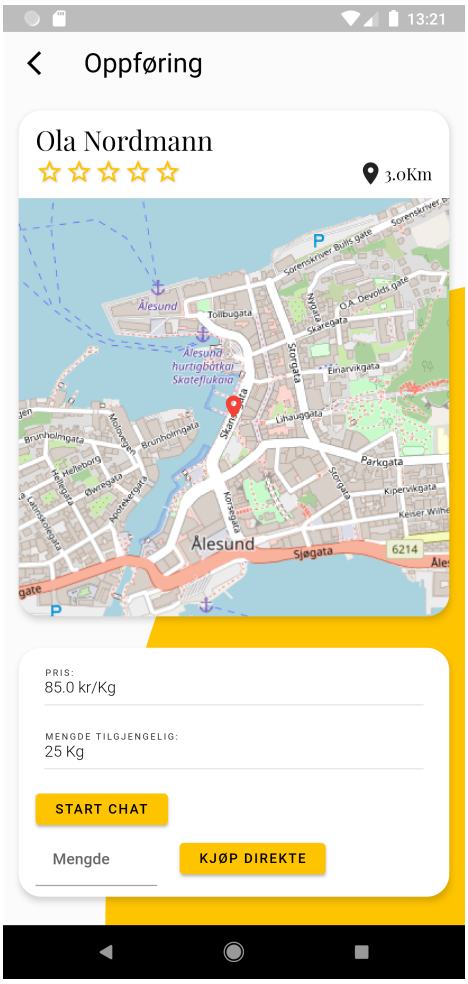
Når man har trykket på en varekategori kan man se gjennom alle annonsene som tilhører denne varekategorien. Siden viser litt informasjon om varekategorien. Hver annonse viser navnet på selgeren, ratingen til selgeren, avstand til "pick-up point", pris på varen og mengde som er gjenværende av varen. Man kan sortere annonsene etter pris, fra lav til høy eller fra høy til lav, med mulighet til å legge til flere sorteringsmetoder i fremtiden. Figur 4.15 viser et skjermbilde av annonsesiden til en varekategori. Når man trykker på en av annonsene blir man tatt til en informasjonsside for den annonsen.



Figur 4.15: Side med alle annonser i en varekategori

4.4.2.3 Annonseinfo

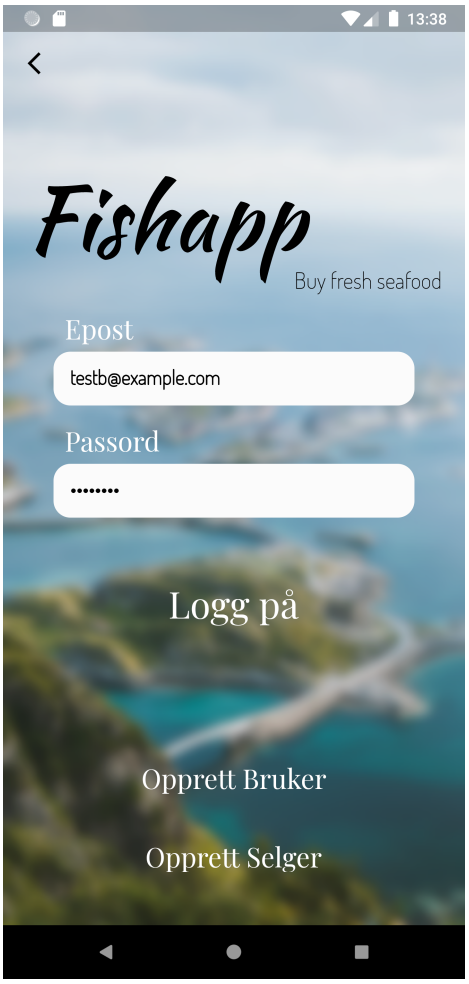
Nå man har trykket på en annonse kommen man inn på informasjonssiden til den annonsen. På denne siden får man se navnet til selgeren, ratingen til selgeren, avstanden til "pick-up point", og et kartsom viser hvor "pick-up point" er. Siden viser også prisen på varen og mengden som er tilgjengelig. På siden er det også to knapper. Den ene er for å starte en chat med selgeren for å f.eks. få mer informasjon om produktet. Den andre er for å kjøpe produktet. Begge disse knappene krever at man har logget seg inn. Figur 4.16 viser et skjermbilde av informasjonssiden til en annonse.



Figur 4.16: Informasjonsside til en annonse

4.4.2.4 Loginn

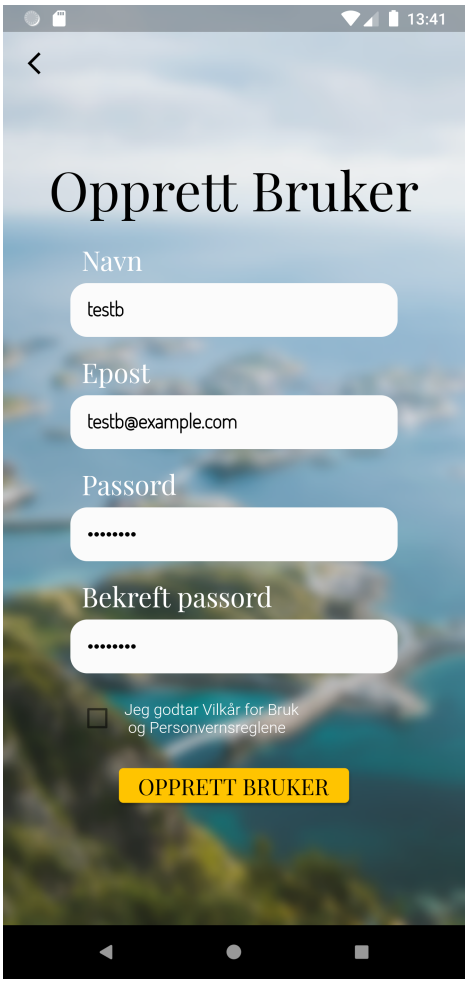
Logg inn siden blir man sendt til hvis man prøver å gjøre noe som krever innlogging og man er ikke logget inn fra før. På denne siden skriver mann inn e-post adressen og passordet til brukeren man har registrert og kan så trykke på *Logg på*. Hvis man ikke har en bruker fra før finnes det to muligheter. Hvis man bare skal ha en vanlig bruker for å kunne kjøpe varer, kan man trykke på *Opprett Bruker* for å bli tatt til siden for å opprette en vanlig bruker. Hvis man er en fisker som skal selge varer i applikasjonen må man trykke på *Opprett Selger*. Da blir man tatt til siden for å opprette selgere. Figur 4.17 viser et skjermbilde av logg inn-siden i mobilapplikasjonen.



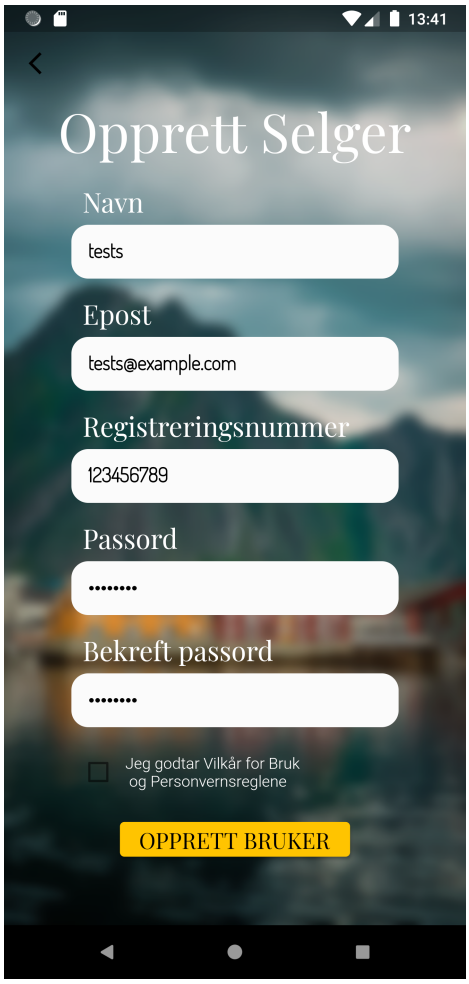
Figur 4.17: Loginn-siden for mobilapplikasjonen

4.4.2.5 Opprette bruker

Det finnes to sider for å opprette brukere, en for vanlige brukere og en for selgere. Bakgrunnen på de to sidene er forskjellige for å enkelt kunne differensiere dem. Begge sidene har felt for navn, e-post adresse, passord og bekreft passord. Opprett selger har i tillegg et felt for et registreringsnummer som fiskere får utdelt fra fiskesalgslag. Dette nummeret viser at de er godkjent til å selge fisk fra båten sin. Så må man krysse av for at man godtar kjøpsvilkårene. Deretter kan man trykke på *Opprett Bruker*-knappen. Hvis alt er i orden blir brukeren opprettet og man blir automatisk logget inn og sendt tilbake til siden man var på før. Figurene 4.18 og 4.19 hvis henholdsvis skjermbilder av sidene for å opprette brukere og selgere.



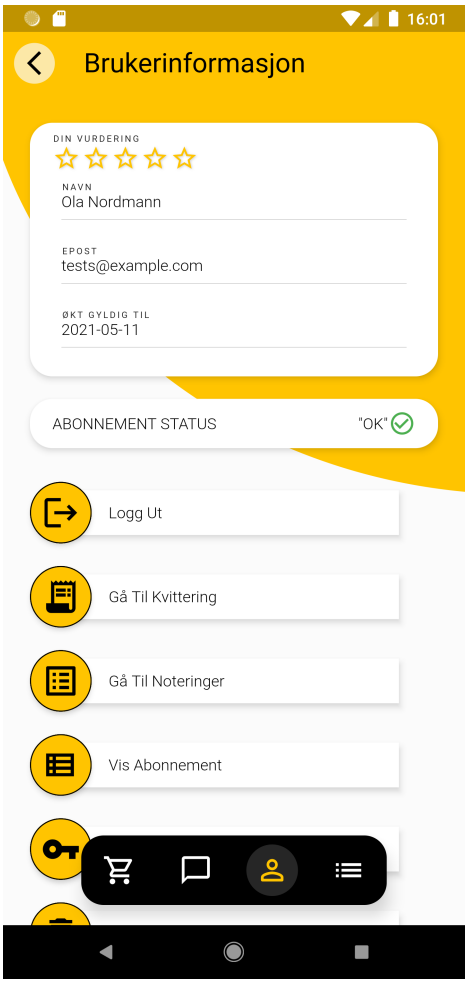
Figur 4.18: Side for å opprette en bruker



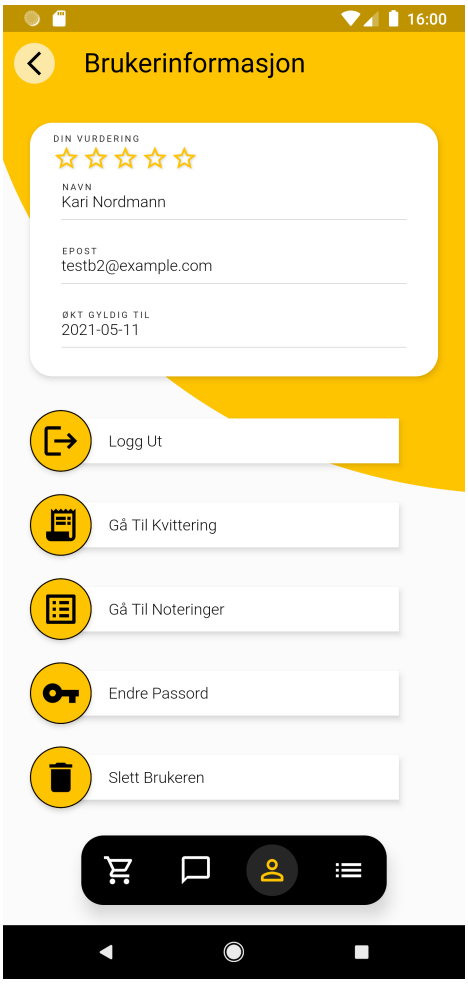
Figur 4.19: Side for å opprette en selger

4.4.2.6 Brukerinformasjon

Siden med brukerinformasjon nås ved å bruke navigasjonsbaren nederst på flere av sidene. På denne siden får man informasjon om profilen sin. Man kan se sin egen rating, navn, epost, og hvor lenge det er til man må logge inn igjen. Det er en knapp som tar deg til en oversikt over alle kvitteringene dine og en annen som tar deg til en oversikt over alle annonsene man har laget. Man har også mulighet til å endre passordet sitt fra denne siden. På denne siden har man og muligheten til å logge ut og slette brukeren sin. Sletting av brukeren gjøres i henhold til GDPR 2.12.1. På siden for kjøpere vil man også kunne se status på abonnementet sitt og det er en ekstra knapp for å se mer informasjon om abonnementet. Figurene 4.20 og 4.21 viser henholdsvis brukerinformasjonssidene for selger og kjøper.



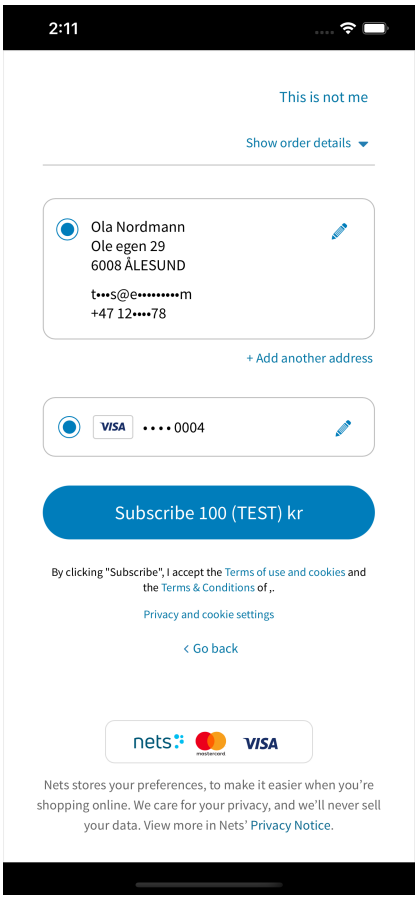
Figur 4.20: Side for brukerinformasjon for selger



Figur 4.21: Side for brukerinformasjon for kjøper

4.4.2.7 Betalingsløsning i applikasjon

Fra siden med abonnement informasjon kan man starte ett nytt abonnement. Når man gjør det, åpnes det et *WebView* som viser en nettside fra Nets. På denne nettsiden skriver man inn personinformasjon og legger til et bankkort som man skal betale med. Denne informasjonen sendes til Nets og om den blir godkjent blir abonnementet opprettet og brukeren tas tilbake til applikasjonen. Figur 4.22 viser et *WebView* hvor det er langt inn betalingsinformasjon.

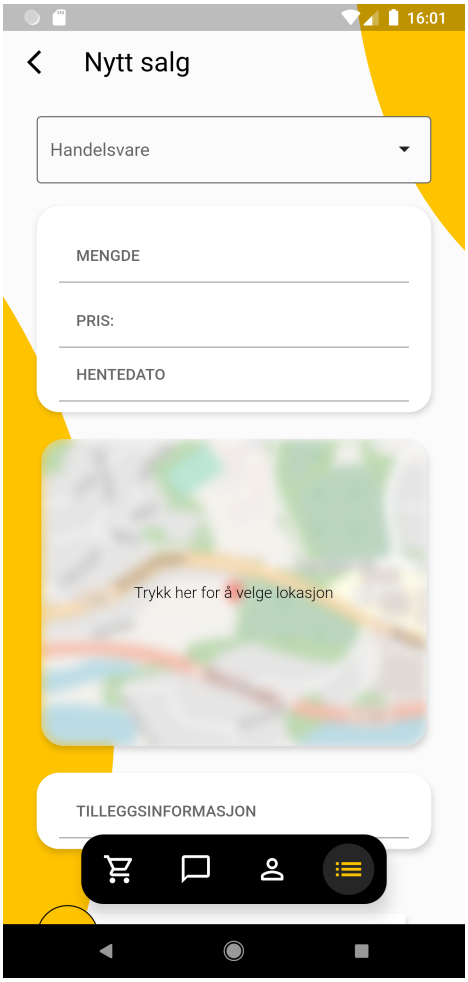


Figur 4.22: Webview for betaling av abonnement

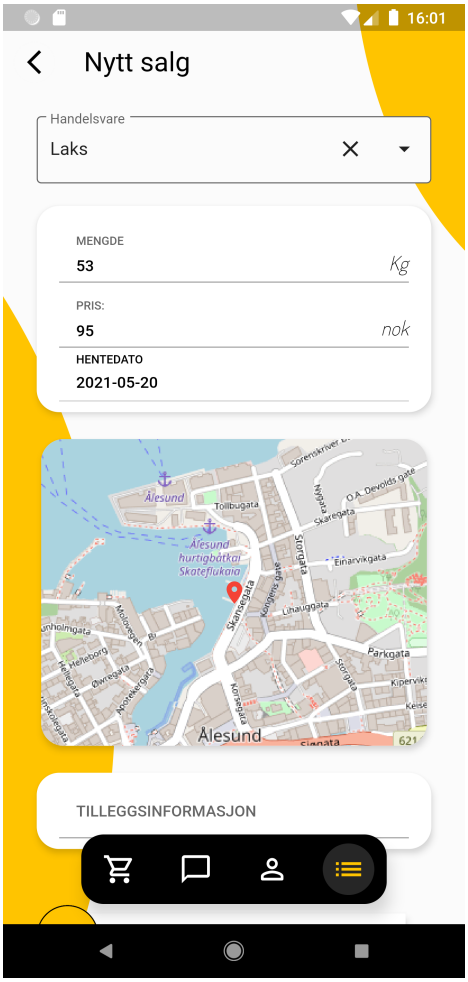
4.4.2.8 Opprette salgs annonse

Som selger kan man opprette en ny salgs annonse. Man kommer til den siden ved å bruke navigasjonsbaren man finner nederst på hovedsiden og flere andre sider. På siden for å opprette en ny salgs annonse er det en nedtrekks meny som er laget ved hjelp av Flutter pakken *dropdown_search* 3.14.2.10. I nedtrekks menyen kan man velge hvilken varekategori det er man skal selge og man kan søke etter en spesifikk varekategori. Videre er det felter for hvor mye av en vare man har å selge, hva prisen skal være, og når kunder kan komme og hente varen. Når man velger hentedato kommer det opp en kalender som man velger dato fra. For å velge hvor hentestedet er trykker man på kartet. Da tas man til en ny side med ett kart man kan navigere og trykke på for å velge lokasjon. Man blir så sent tilbake til siden for å opprette en annonse som nå viser den valgte lokasjonen. Det er også et felt for tilleggsinformasjon hvis man vil legge til noe. Så kan man opprette annonsen ved å trykke på knappen nederst på siden. Hvis alle feltene er fylt ut korrekt blir annonsen opprettet og man blir tatt til informa-

sjonssiden til annonsen. Hvis noen av feltene er fylt ut feil får man beskjed om det. Figurene 4.23 og 4.24 viser henholdsvis siden før man har lagt til noen informasjon eller valgt lokasjon og etter man har lagt til all informasjonen.



Figur 4.23: Side for nytt salg før man velger lokasjon



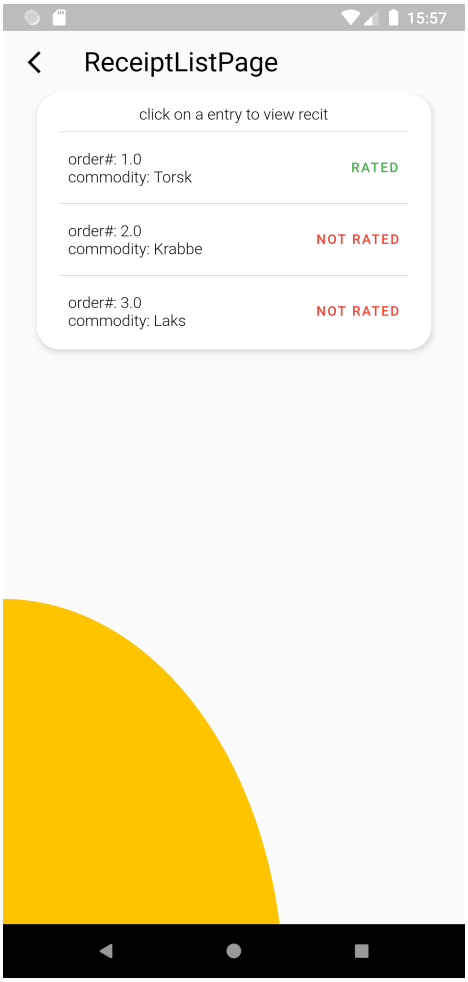
Figur 4.24: Side for nytt salg etter man har valgt lokasjon

4.4.2.9 Gjennomføre kjøp

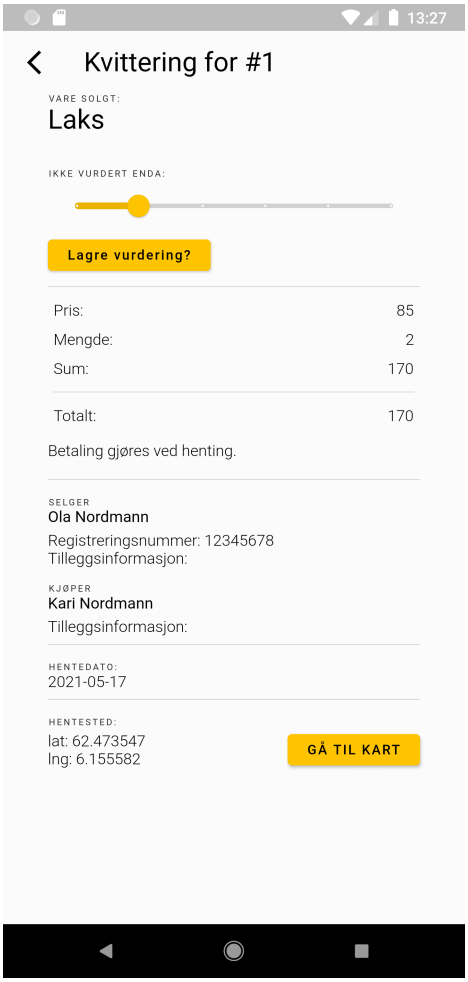
På siden med info om en annonse 4.4.2.3 har man mulighet til å kjøpe en mengde av varen som blir solgt. Det er et felt hvor man fyller ut hvor mye av varen man ønsker å kjøpe og så trykker man på *Kjøp*-knappen. Da får man opp en kvittering hvor man får se hvor mye man trenger å betale. Betaling gjøres direkte med selgeren når man henter varen.

4.4.2.10 Kvitteringer

Etter man har gjennomført et kjøp eller salg får man en kvittering for handelen. Fra siden med brukerinformasjon får man tilgang til en liste over alle kvitteringer som man har i applikasjonen. Denne listen viser hva man har kjøpt og om man har ratet motparten eller ikke. Hvis man trykker på en av kvitteringene i listen kommer man inn på en side som viser all informasjonen i kvitteringen. Her kan man rate motparten om man ikke har gjort det. Kvitteringen viser hva man har kjøpt, hva varen kostet, hvor mye av varen man kjøpte, og summen av det. Man ser også hvem man kjøpte varen av og når man kan hente varen. Det vises også hvor man kan hente varen med mulighet til å få se det i en ekstern kartapplikasjon som Google Maps. Figurene 4.25 og 4.26 viser henholdsvis listen over kvitteringer i applikasjonen og informasjonssiden til en kvittering.



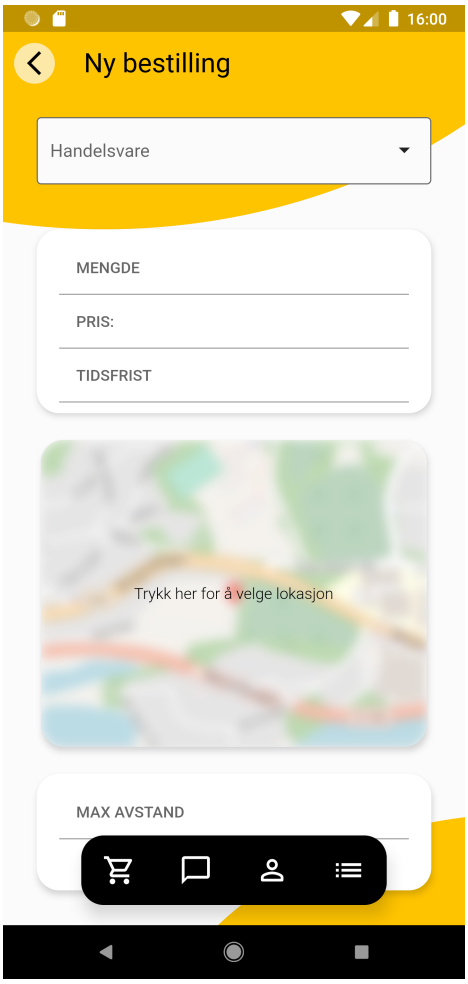
Figur 4.25: Liste over kvitteringer



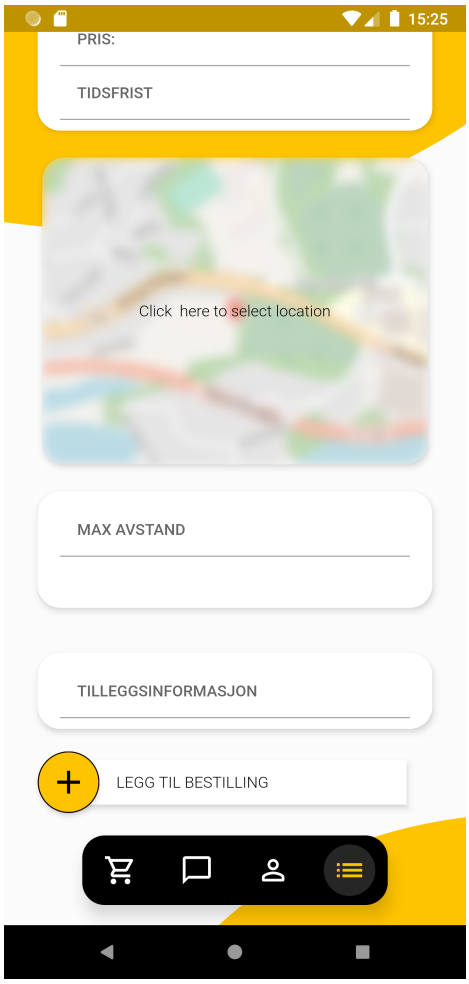
Figur 4.26: En kvittering i applikasjonen

4.4.2.11 Opprette bestilling

Som en vanlig bruker har man muligheten til å opprette en bestilling for en vare man ønsker å kjøpe. Man når denne siden ved å bruke navigasjonsbaren nederst på hovedsiden og flere andre sider. Denne siden er veldig lik den selgere bruker for å opprette salg. Den har en nedtrekks meny for å velge varen man ønsker å kjøpe, et felt for ønsket mengde, et felt for ønsket pris, og et felt for fristen man ønsker bestillingen fullført innen. Siden har også ett kart hvor man velger sin lokasjon og under skriver man maksimal avstand man ønsker å reise for å hente varen. Begge disse er for at fiskere bare kan fullføre bestillinger som ikke for langt unna området de selger fisk i. Til slutt er det felt hvor man kan legge til tilleggsinformasjon. Når man har fylt ut alle feltene, trykker man på knappen *Legg til bestilling*. Hvis feltene er fylt korrekt, blir bestillingen opprettet og man blir tatt til informasjonssiden for bestillingen sin. Hvis noen av feltene er fylt ut feil får man beskjed om det. Figurene 4.27 og 4.28 viser hele siden for nye bestillinger med alle feltene.



Figur 4.27: Toppen av siden for å opprette ny kjøpsordre



Figur 4.28: Bunnen av siden for å opprette ny kjøpsordre

4.4.2.12 Støtte for flere språk

Ett av kravene i oppgaven var at applikasjonen skulle ha støtte for både Norsk og Engelsk språk. Ved programvareutvikling er slik funksjonalitet kjent under faguttrykket lokalisering 2.6.2. Flutter har god støtte for slik funksjonalitet ved hjelp av pakken *intl* 3.14.2.15. For å implementere dette på en effektiv måte, ble det besluttet i starten av utviklingsprosessen at all tekst i brukergrensesnittet som hovedregel skulle støtte lokalisering.

4.4.2.13 Kart og GPS

Ett annet av kravene for applikasjonen var GPS- og kartintegrasjon for å kunne sette "pick-up-point" og vise dette til brukerne. For å få denne funksjonaliteten var det flere pakker vi

kunne bruke, men vi valgte å gå for *Geolocator* 3.14.2.8 og *Flutter Map* 3.14.2.7. Ved å bruke både Geolocator og Flutter Map kan vi vise avstand fra posisjonen til bruker til "pick-up-points" satt av selgere. Vi fikk også lagt til at selgere kan velge posisjonen til "pick-up-point"-et ved å trykke på et interagerbart kart. Ved å bruke *Map Launcher* 3.14.2.5 har brukerne mulighet til å få veivisning til et "pick-up-point" i en ekstern kart applikasjon.

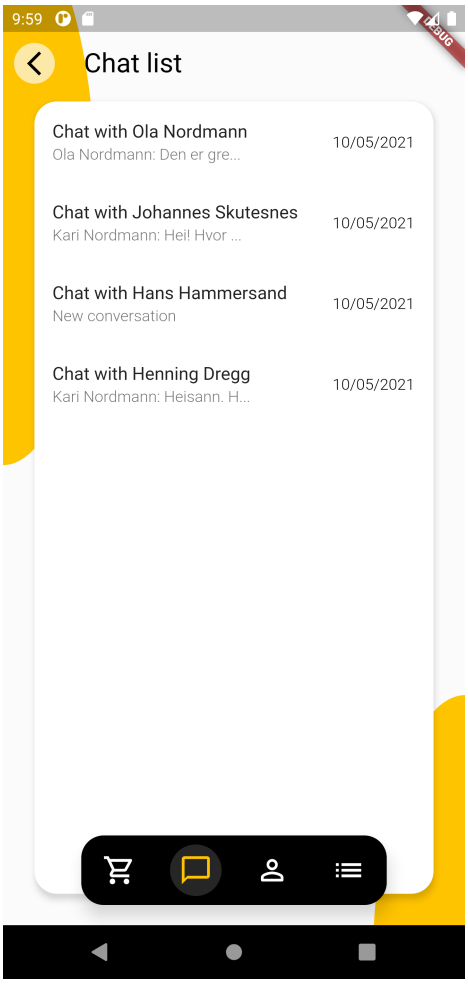
4.4.2.14 Rangering av selger

I applikasjonen kan kjøpere og selgere rangeres med en score på 1-5. Når et kjøp er gjennomført får både kjøper og selger mulighet til å rangere motparten. Den gjennomsnittlige scoren til en bruker vil vises på profilen deres og vil også vises på oppføringer laget av den brukeren. På denne måten vil brukere gjøre et mer informert valg av hvem de vil kjøpe fra. En slik rangering vil også oppfordre til bedre kundeservice fra selgerne og god oppførsel fra kjøperne. 4.3.5.3

4.4.2.15 Chat mellom brukerne

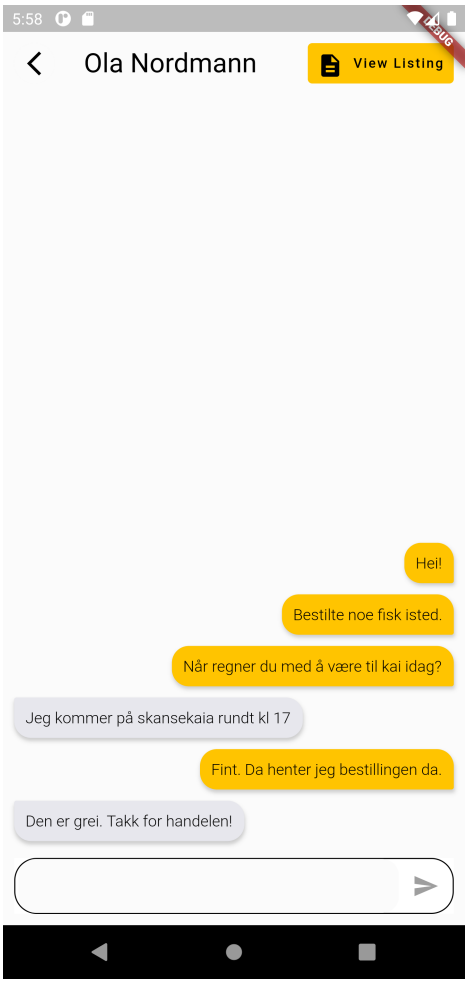
En forutsetning for å gjennomføre en handel, er a sørge for god kommunikasjon mellom kjøpere og selgere. Ut ifra krav i oppgaveteksten og i samråd med oppdragsgiver ble det klart at det ville være avgjørende at mobilapplikasjonen har funksjonalitet for å chatte mellom brukerne.

I applikasjonen er det en egen seksjon satt av til dette, som kommuniserer med et eget API mot *Chat-komponenten* på serveren. Figur 4.29 viser liste over samtaler. Samtaler er knyttet mot en salgsannonse eller kjøpsforespørsel - slik at brukerne enkelt kan se hva samtalen gjelder. En samtale i applikasjonen kan opprettes på to måter; fra en salgsannonse (som kjøper) eller fra en kjøpsforespørsel (som selger). Når en selger har laget en kjøpsannonse, kan kjøpere via annonsen opprette en samtale med selgeren. På motsatt vis kan kjøpere publisere en kjøpsforespørsel, der selgere kan opprette en samtale med kjøperen.

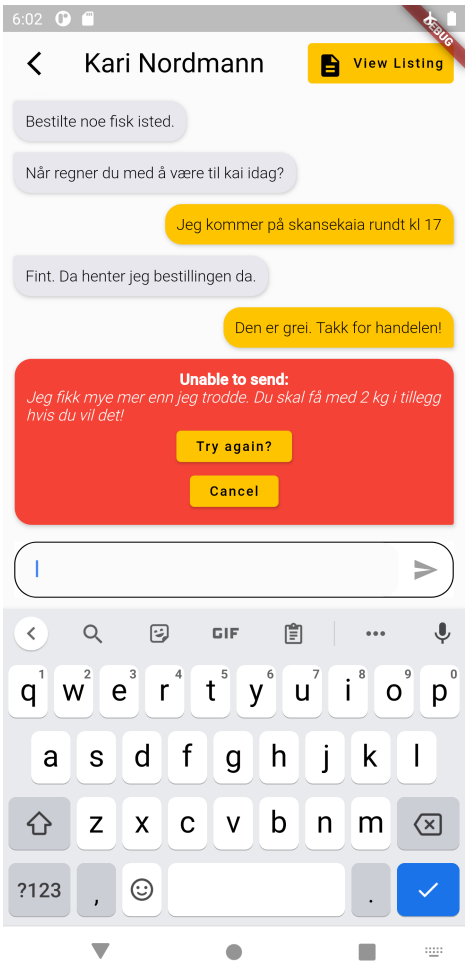


Figur 4.29: Skjerm som viser oversikt over alle samtaler en bruker har

Samtaler som brukerne er knyttet til, er synlig i en liste i mobilapplikasjonen, dette vises på Figur 4.29. Hver av brukerne kan gå inn i den respektive samtalen for å lese eller sende meldinger. For å sørge for at brukerne alltid kan se de siste meldingene, er det satt opp en *polling-mekanisme*; denne sjekker regelmessig med server om det er kommet nye meldinger i samtalen.



Figur 4.30: Kjøper har samtale med selger



Figur 4.31: Selger mistet Internett før svar

Polling-mekanismen er implementert ved at bare nødvendige meldinger overføres. Ved spørring mot server, vil mobilapplikasjonen bare motta meldinger den ikke har lastet ned fra før. Dette fungerer ved at applikasjonen i en *modell* lagrer *metadata*; opplysninger som omhandler selve samtalen, som t.d. dato, bruker som opprettet samtalen, tilknyttede *listing* (kjøpsforespørsel/vareannonse for motpart). Samtale-skjermen vises i Figur 4.30 og 4.31.

Hver av meldingene i en samtale er nummererte. Modellen i mobilapplikasjonen lagrer til enhver tid en liste over meldinger, inkludert nummer på hver melding. Dette nummeret er et sekvenstall, og er en del av *metadata* som oppdateres hver gang applikasjonen kommuniserer med API-et på server. Ved sjekking etter nye meldinger, vil mobilapplikasjonen sammenligne sekvenstallet for siste melding fra lokal modell mot *metadataen* den fikk fra server. Ved spørring etter nye meldinger mot server, vil mobilapplikasjonen sende sekvenstallet på siste mottatte melding. Chat-komponenten på server vil motta dette og kalkulere ut meng-

den manglede meldinger. Hvordan dette gjøres i detalj, forklares på 4.3.5.3.

Fordi datamengden for hver av spørringene er liten, muliggjør dette at intervallet mellom hver spørring kan settes lavt. Under testing har vi opplevd gode erfaringer med et intervall på 5 sekunder. Dette gjør at i realiteten oppleves det for brukerne at meldingene overføres i sanntid. Det kan hende i en produksjonssammenheng med flere brukere, at intervalltiden må justeres til en lengre periode.

4.4.3 Kommunikasjon med server

For at mobilapplikasjonen skal fungere er den avhengig av å kunne kommunisere med serveren. For å løse dette har vi implementert REST API-grensesnitt 2.9.4, som benytter HTTP 2.9.3 til overførsel av data. Denne overførselen skjer ved hjelp av TCP/IP 2.9.1 over Internett. Under testing av mobilapplikasjonen har vi satt opp en test server ved skolens servermiljø. For å enkelt nå tak i serveren, er det også satt opp ett domenenavn, med en DNS-peker 2.9.2 som leder til serveren IP-adresse. Hvis IP-adressen for serveren skulle endre seg, kan vi enkelt sette en ny DNS-peker, istedenfor å måtte bygge mobilapplikasjonen på nytt. Siden kommunikasjonen foregår offentlig via mange datanettverk og noder, benytter vi TLS 2.11.1.1 som et sikkerhetstiltak. TLS sørger for at informasjon som blir sendt er kryptert. Fordi vi hadde satt opp serveren med domenenavn, og korrekt DNS-peker, kunne vi enkelt ta i bruk tjenesten Let's Encrypt 3.7.5.2 til å signere sertifikatet vårt.

4.4.4 Sikkerhet

For sikkerhet på mobilapplikasjonen, fokuserte vi på at autentiseringsnøkkelen fra serveren kunne lagres sikkert og ikke leses av andre applikasjoner på telefonen.

4.4.4.1 Håndtering av JWT

Når en bruker utfører en innlogging i applikasjonen, vil vedkommende få tilsendt en autentiseringsnøkkel fra serveren. Nøkkelen av typen JWT 3.7.4.8, representerer identiteten til en bruker. Denne benyttes i situasjoner der det er nødvendig å fastslå hvem brukeren er, for ek-

sempel ved utførelse av en handel. Av den grunn betraktes nøkkelen som sensitiv data, og må lagres på en slik måte at ikke andre kan få tilgang til den. Dette ble løst ved å ta i bruk pakken *flutter_secure_storage* 3.14.2.12, som implementerer et API for å lagre data kryptert på både iOS og Android -plattformene. Kodeeksempel 4.18 viser et utdrag fra *appstate.dart* på hvordan dette er håndtert.

Kodeeksempel 4.18: Lagring av JWT til kryptert lager *(appstate.dart)*

```
1  class AppState extends ChangeNotifier{
2      ...
3      void newAuthValues(String token, User user) {
4          jwtTokenData = JwtTokenData.buildFromTokenStr(token);
5          this.user = user;
6          SecureStorage().writeTokenString(token);
7          SharedStorage().saveUser(user);
8          notifyListeners();
9      }
10     ...
11       void loggOut() {
12     SecureStorage().removeTokenString();
13     SharedStorage().removeUser();
14     this.user = null;
15     this.jwtTokenData = null;
16     notifyListeners();
17   }
18 }
```

Del 5

Drøfting

I denne delen drøfter vi ulike deler av det arbeidet vi har gjort gjennom prosjektet. Først går vi gjennom tankene våre rundt bytte av arkitektur vi gjorde på applikasjonsserver. Så drøfter vi resultatene, og deretter drøfter vi selve gjennomføringen av prosjektet.

5.1 Bytte av applikasjonsserver

I denne seksjonen reflekterer vi rundt bytte av applikasjonsserver i prosjektet. Først beskriver vi bakgrunnen for dette bytte, deretter går vi gjennom våre erfaringer ved bytte av applikasjonsserveren, overgang fra monolitt til mikrotjenester og bruk av Kubernetes. Til slutt vil vi gjøre noen refleksjoner om dette bytte var et klokt valg.

5.1.1 Bakgrunn

Vi hadde initialt utviklet en serverapplikasjon med en monolittisk arkitektur, som kjørte på applikasjonsserveren Payara. Til denne utviklet vi en mobilapplikasjon. Omtrent ved påsketider var denne server-app kombinasjonen ferdig. Det eneste som manglet var betalingsløsningen, hvor vi avventet svar fra leverandør. Prosjektgruppen så da at vi hadde en god mulighet til å utforske og prøve en ny serverarkitektur. Denne idéen stammet fra at flere av gruppemedlemene fant serverarkitekturer interessant, spesielt mikrotjenester.

Fordi vi allerede hadde en monolittisk implementasjon av løsningen, ville vi også få muligheten til å sammenlikne disse arkitekturene. Men med en knapp måned gjenstående, ville dette bli en utfordring, spesielt siden ingen på gruppen hadde noen tidligere erfaring med verken Kubernetes, MicroProfile eller mikrotjenester.

5.1.2 Bytte av applikasjonsserver

Etter gruppen hadde blitt enige om å implementere mikrotjenester gjenstod spørsmålet om vi skulle beholde applikasjonsserveren vi hadde. Serveren vi startet med var Payara3.7.4.4. Vi valgte denne siden flere av gruppemedlemmene hadde erfaring med serveren fra tidligere. Etter litt undersøkelse fant gruppen OpenLiberty 3.7.4.5 som et alternativ. Denne applikasjonsserveren følger også JakartaEE spesifikasjonen 3.7.1 men har 2 fordeler ovenfor Payara. Det første er at vi fant dokumentasjonen og eksemplene (IBM 2021c) til OpenLiberty er usedvanlig gode. Det andre er at OpenLiberty støtter hot-reloding, noe vi anså ville spare mye tid under utvikling. Gruppen fant senere ut at denne featuren hadde flere problemer i mere komplekse prosjekter, og vi endte opp med heller å lage script for og re-bygge og redeploye containerene manuelt.

Verdien av å programmere opp mot en spesifikasjon i motsetning til en spesifikk server, ble klart tydelig i overgangen fra Payara til OpenLiberty. Mesteparten av kodebasen kunne flyttes over med minimale modifikasjoner. I en slik prosess blir det også tydelig hva som ikke er dekt av spesifikasjonen. Dette erfarte vi med multipart form hvor vi måtte implementere en egen handler for bruke IBM sin spesifikke implementasjon av dette (IBM 2021b) beskrevet i 4.3.3.8.

Prosjektgruppen opplevde vi at denne overgangen gikk veldig fint.

5.1.3 Overgang til mikrotjenester

Den største utfordringen med bytte av backend var overgangen fra monolitt struktur til mikrotjenester. Det første vi måtte vurdere var hvordan vi skulle splitte opp monoliten. Hvordan vi endte opp å splitte er beskrevet i 4.3.3. Selve prosessen med å splitte ut modulene gikk overraskende enkelt for seg. Dette er antakelig vis fordi serveren allerede hadde godt kodedesign

med høy konsistens 2.2.1 og lav kobling 2.2.2. Prosjektgruppen fant det høyst interessant å se et så klart og tydelig eksempel på hvor nyttig godt kodedesign kan være, og hvor godt det merkes når kodekvaliteten i en av modulene ikke er helt den samme.

For å forenkle prosessen valgte vi å følge bruke MicroProfile spesifikasjonen 3.7.2. Denne var veldig nyttig, siden den løste mange av de vanlige problemene man møter ved utvikling av mikrotjenester. Foreksempel statussjekk av tjenestene beskrevet i 4.3.4.2 og kommunikasjon mellom tjenestene beskrevet i 4.3.3.8.

Splittelsen av monolitten ledet også til splittelse av databasen. Dette skapte uforutsette konsekvenser for mobilapplikasjonen 5.1.6.

5.1.4 Refleksjoner rundt mikrotjenester eller monolitt

Gjennomføringen av byttet fra monolitt til mikrotjeneste har vært interessant. Og selv om prosjektgruppen trygt kan konkludere med at mikrotjenester ikke er gunstig i prosjekter av denne størrelsen, ser vi noen positive og negative sider ved mikrotjenester. Disse beskriver vi i denne delen.

5.1.4.1 Positive sider ved mikrotjenester

Mye av fordelene med mikrotjenester, ligger i fleksibiliteten denne arkitekturen tilbyr. Muligheten til å bryte store monolitter opp i mindre, mer håndterlige deler, gir utviklere muligheten til å gradvis rulle ut endringer eller nye teknologier.

De fleste av problemene mikrotjenester løser kommer ved økende kompleksitet og skala i software prosjekter. Dette prosjektet var relativt sett ganske lite. Derfor erfarte vi at at mye av problemene mikrotjenester løser var problemer vi enda ikke hadde begynt å få. Over tid, om prosjektet skulle øke i størrelse, kunne en slik mikrotjeneste arkitektur gi fremtidige utviklere mye nytte.

5.1.4.2 Negative sider ved mikrotjenester

Alle mikro-tjeneste applikasjoner er i sin kjerne en monolitt, der funksjonskallene har blitt erstattet med nettverkskall og databasen har blitt splittet i mindre biter. For å kompansere for disse endringene vil det alltid kreves en viss mengde logikk. Hva man får i gjengjeld, er frittstående komponenter med lavere minneavtrykk som kan skaleres og kjøre individuelt. Siden mange av funksjonskallene har blitt erstattet med http kall vil det også føre til en ytelses kostnad.

En plass hvor vi erfarte økt kompleksitet, var i sammenheng med splittelsen av databasen. Dette medførte forandringer i datastrukturen, som videre krevde at mobilapplikasjonen måtte endres.

5.1.5 Kubernetes

Overgangen fra Docker Compose til bruk av Kubernetes var interessant. Prosjektgruppen fikk kjapt et inntrykk av hvor omfattende dette verktøyet er, og fant det veldig interessant å utforske og fordype seg i. Noe vi fikk erfare med Kubernetes, var at selv om det kan være vanskelig å forstå, så er det relativt lett å ta i bruk når det er satt opp.

Våre erfaringer med Kubernetes har vært gode og nyttige. Til tross for den økte kompleksiteten, har det vist seg å være et svært praktisk redskap for å lage nye mikrotjenester.

5.1.6 Resultat

Det at vi fikk muligheten til å prøve ut både monolitt og mikrotjenester på samme prosjekt, har vært en veldig lærerik og interessant opplevelse. Det som vi erfarte, var at det ikke er noen mirakel-arkitektur som løser alle problemer. Det er fordeler og bakdeler med både monolitt og mikrotjeneste og hva som er rett og bruke vil avhenge av prosjekt og situasjon.

5.2 Mobilapplikasjon

I denne seksjonen diskuterer vi ulike aspekter ved mobilapplikasjonen som vi utviklet gjennom prosjektet.

5.2.1 Flutter som rammeverk

Prosjektgruppen er fornøyd med den produserte mobilapplikasjonen. Vi var overrasket over hvor mye enklere og kjappere vi kunne utvikle applikasjoner i Flutter rammeverket, sammenliknet med andre rammeverk vi er kjente med, som Java. Det kan tenkes at dette stammer fra friheten Google får, som utvikler av rammeverk og språk.

5.3 Server

I denne delen vil vi diskute de forskjellige aspektene med distribuere og konfigurering av systemet.

5.3.1 Deployering av systemet

Prosjektgruppen har gjennom prosjektet hatt stort fokus på å utvikle i et miljø som er så "Close to production" som mulig. Vi fikk erfare mot slutten av prosjektet hvor gunstig dette valget var, siden vi lett kunne installere systemet på en server ved NTNU i Ålesund.

5.3.2 Konfigurering av systemet

At serveren kan konfigureres ved å endre ett sett med *env*-filer er gruppen veldig fornøyde med. Siden vi hadde tidlig fokus på å ha et strømlinjeformet konfigurasjons oppsett har vi spart mye tid på at konfigurasjonsendringer ikke trengs å dupliseres, og derav færre "config-out-of-sync" bugs. Konfigurasjonsoppsettet kan også lett endres over til å bruke *Kubernetes-secrets*, siden OpenLiberty har et konfigurasjonshierarki.

5.4 Kommunikasjon

Her beskriver vi hvordan kommunikasjonen med oppdragsgiver og innad i gruppen har gått gjennom hele prosjektet.

5.4.1 Oppdragsgiver

Kommunikasjonen med oppdragsgiver har foregått veldig bra. Oppdragsgiver arbeider ombord på en fiskebåt og er ute på havet i flere uker om gangen. Kommunikasjonen har derfor gått for det meste over e-post. Til tross for dette synes vi at kommunikasjonen har vært veldig bra. Vi har sendt oppdragsgiver planene våre for design på mobilapplikasjonen og har fått gode tilbakemeldinger på disse. Vi har også regelmessig sendt oppdragsgiver e-post om den foreløpige planen vår og hvordan fremdriften har vært. Oppdragsgiver har vært veldig flink med å gi konstruktive tilbakemeldinger over e-post så denne kommunikasjonsformen har ikke vært ett stort hinder.

Mens oppdragsgiver var i land, hadde vi to fysiske møter med dem. På det første møtet gikk vi mer grundig over designet på mobilapplikasjonen og vi fikk gode tilbakemeldinger fra oppdragsgiver som hjalp oss videre i prosjektet. Vi fikk også diskutert mer grundig med oppdragsgiver hvordan han ønsket betalingsløsningen. På det andre møtet viste vi frem mobilapplikasjonen som på det tidspunktet var stort sett ferdig. Oppdragsgiver var da veldig fornøyd med den.

Vi opplever at kommunikasjon via e post har fungert godt, men ser allikevel at fysiske møter har vært nyttig som et supplement. I en ideell situasjon kunne nok flere fysiske møter vært ønskelig. På de fysiske møtene har vi kunnet diskutere mobilapplikasjonen med tegninger og figurer slik at både vi og oppdragsgiver har hatt en bedre forståelse om hvordan slutt produktet skulle se ut.

5.4.2 Innad i gruppen

I starten av prosjektet bestemte gruppen seg for at vi skal ha standup møte 3.3.3 hver dag klokken 09.00. Det har vi klart å overholde, og det har bidratt til god struktur i hverdagen.

Vi mener den gode strukturen har bidratt til stor produktivitet Møtet var et stand – up møte på ca 15 minutter hvor hver enkelt redegjorde for sitt arbeide, for å sikre en felles forståelse for arbeidet videre. Når noen hadde utfordringer hadde vi et møte hvor vi i fellesskap fant løsninger på utfordringene. Vi holdt møtene gjennom "voice chat" funksjonen til Discord 3.2.5.1. Vi brukte også Discord for å kommunisere via tekst, og brukte denne funksjonen for å stille spørsmål og spørre om hjelp. Discord har vært et viktig verktøy for vår kommunikasjon og dette har sørget for at vi har hatt utmerket kommunikasjon innad i gruppen.

Gjennom hele prosjektet har gruppen jobbet på hjemmekontor. Dette har hatt både fordeler og ulemper. Ulempene har vært at vi mistet den kontakten vi ville ha hatt om vi jobbet i samme rom. Vi mistet muligheten til å enkelt spørre de andre om hjelp og kjapt se over koden sammen. Om vi jobbet i samme rom kunne vi enkelt vist problemer til hverandre og tenkt ut og prøvd løsninger. Vi har løst dette som nevnt tidligere ved å kommunisere via Discord. Fordelene har vært at gruppen har jobbet mer effektivt, på grunn av færre forstyrrelser som ville oppstått om vi jobbet sammen i samme rom.

5.5 Gjennomføring

I denne seksjonen har vi tatt for oss selve gjennomføringen av prosjektet og hvordan det gikk.

5.5.1 Utviklings metodikk

Vi har gjennom prosjektet brukt metodikken Scrum. Vi har hele tiden overholdt sprintlengden på 2 uker, og har hatt stand-up møte hver ukedag. Etter hver sprint har vi gått gjennom progresjonen vi hadde, om arbeidet hadde vært bra , og hva vi kunne forbedre. Vi hadde også et møte med veilederen vår ved slutten av hver sprint. Møtene med veileder har vært nyttig da vi opplever å ha fått konstruktive og nyttige tilbakemeldinger på det arbeide vi hadde utført. Rapportene fra sprintene finnes i vedlegg F.

Det har fungert veldig bra å gjennomføre prosjektet på denne måten, siden vi kontinuerlig har kunnet vurdere hvor langt vi har kommet og hvilke arbeidoppgaver som måtte prioriteres.

Del 6

Konklusjon

Vi kan konkludere med at denne oppgaven har vært en svært lærerik opplevelse. Denne muligheten til å opparbeide oss erfaring med redskaper og teknologier som brukes i industrien i dag, tenker gruppen vi vil ha nytte av i yrkeslivet uavhengig av om dette blir i industrien eller annen type arbeid som dataingeniør.

Det har vært en interessant og utfordrende prosess å bygge en løsning som strekker seg over mange forskjellige felt. Prosjektgruppen mener løsningen dekker godt behovet presentert i problemstillingen oppdragsgiver beskrev for oss. Vi har bygget en mobilapplikasjon i Flutter som fungerer på tvers av forskjellige operativsystem. Vi har også utviklet et webgrensesnitt i det moderne rammeverket React for å administrere denne applikasjonen. Når det kommer til serveren, valgte vi å virkelig utfordre oss selv, ved å bruke en mikrotjenestearkitektur. Dette resulterte i en applikasjonserver som besto av 7 forskjellige OpenLiberty mikrotjenester som vi kjører på Kubernetes.

Vi hadde initialt utviklet en fulstendig applikasjons server med monolittisk arkitektur, før vi senere gikk over til mikrotjenester. Dette gav oss den unike muligheten til å få innsikt i fordelene og ulempene med disse to forskjellige arkitekturene.

Siden vi i praksis har utviklet to fullstendige applikasjonsservere, står fremtidige utviklere fritt om hvilken implementasjon de ønsker å bygge videre på.

Bibliografi

Adobe (2021). *XD Individual Pricing | Free Starter Plan Info | Adobe XD*. en. URL: https://www.adobe.com/products/xd/pricing/individual.html (sjekket 03.03.2021).

Agile Alliance (29. jun. 2015). *What is Agile Software Development?* Agile Alliance |. URL: https://www.agilealliance.org/agile101/ (sjekket 22.04.2021).

Amazon Web Services (24. apr. 2021). *What is DevOps? - Amazon Web Services (AWS)*. [Online; accessed 18. May 2021]. URL: https://aws.amazon.com/devops/what-is-devops.

Atlassian (feb. 2021). *User Stories | Examples and Template*. en. URL: https://www.atlassian.com/agile/project-management/user-stories (sjekket 25.02.2021).

AXELOS ITIL® Publications (okt. 2012). *Glossaries of Terms | AXELOS*. [Online; accessed 19. Apr. 2021]. URL: https://www.axelos.com/glossaries-of-terms.

Axios (mai 2021). [Online; accessed 13. May 2021]. URL: https://axios-http.com (sjekket 13.05.2021).

Barrett, Daniel (2001). *SSH, the secure shell : the definitive guide*. Cambridge Mass: O'Reilly. ISBN: 978-0-596-00011-0.

Bass, Len, Paul Clements og Rick Kazman (9. apr. 2003). *Software Architecture in Practice*. Google-Books-ID: ZY6UZTjBnGQC. Addison-Wesley Professional. 572 s. ISBN: 978-0-321-68041-9.

Beck, Kent mfl. (29. jun. 2015). *Agile Manifesto for Software Development | Agile Alliance*. Agile Alliance |. URL: https://www.agilealliance.org/agile101/the-agile-manifesto/ (sjekket 22.04.2021).

Benevides, Rafael (2019). *Microservices for Java developers : a hands-on introduction to frameworks and containers*. Sebastopol, CA: O'Reilly Media. ISBN: 9781492038283.

Burns, Brendan (2019). *Kubernetes : up and running: dive into the future of infrastructure*. Sebastopol, CA: O'Reilly. ISBN: 9781492046530.

Burns, Brendan mfl. (mar. 2016). *Borg, Omega, and Kubernetes - ACM Queue*. [Online; accessed 30. Apr. 2021]. URL: https://queue.acm.org/detail.cfm?id=2898444#content-comments.

Canonical (11. mai 2021). *MicroK8s - Zero-ops Kubernetes for developers, edge and IoT | MicroK8s*. [Online; accessed 11. May 2021]. URL: https://microk8s.io.

Carlson, Adam (23. apr. 1996). *Coupling and Cohesion*. [Online; accessed 18. May 2021]. URL: https://courses.cs.washington.edu/courses/cse403/96sp/coupling-cohesion.html.

Chang, Donghoon mfl. (2019). «Cryptanalytic time–memory trade-off for password hashing schemes». I: *International journal of information security* 18.2. Place: Berlin/Heidelberg Publisher: Springer Berlin Heidelberg, s. 19. ISSN: 1615-5262.

CloudFlare Inc. (11. mai 2021). *What Is A Reverse Proxy? | Proxy Servers Explained*. [Online; accessed 11. May 2021]. URL: https://www.cloudflare.com/en-gb/learning/cdn/glossary/reverse-proxy.

Configure Liveness, Readiness and Startup Probes (apr. 2021). [Online; accessed 13. May 2021]. URL: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes (sjekket 13.05.2021).

Cross-platform software (7. mai 2021). I: *Wikipedia*. Page Version ID: 1021968616. URL: https://en.wikipedia.org/w/index.php?title=Cross-platform_software&oldid=1021968616 (sjekket 11.05.2021).

Datatilsynet (16. mai 2021). *Ordliste*. [Online; accessed 16. May 2021]. URL: https://www.datatilsynet.no/regelverk-og-verktoy/ordliste.

Den foretrukne betalingspartneren — Nets (mai 2021). [Online; accessed 8. May 2021]. URL: https://www.nets.eu/no/payments (sjekket 08.05.2021).

DIBS Easy E-commerce - DIBS Technical Documentation (mar. 2021). [Online; accessed 8. May 2021]. URL: https://tech.dibspayment.com/easy (sjekket 08.05.2021).

Discord Inc (feb. 2021). *Why Discord Is The Best Place To Talk and Hang Out*. en-US. URL: https://discord.com/why-discord-is-different (sjekket 25.02.2021).

Dragoni, Nicola mfl. (2017). «Microservices: Yesterday, Today, and Tomorrow». I: *Present and Ulterior Software Engineering*. Red. av Manuel Mazzara og Bertrand Meyer. Cham: Springer International Publishing, s. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

Dusseault, Lisa M. og James M. Snell (mar. 2010). *PATCH Method for HTTP*. RFC 5789. DOI: 10.17487/RFC5789. URL: https://rfc-editor.org/rfc/rfc5789.txt.

Eclipse Foundation (feb. 2020). *Configuration for MicroProfile*. [Online; accessed 24. Apr. 2021]. URL: https://download.eclipse.org/microprofile/microprofile-config-1.4/microprofile-config-spec.html.

— (14. mai 2014). *EclipseLink/FAQ/JPA - Eclipsepedia*. URL: https://wiki.eclipse.org/EclipseLink/FAQ/JPA (sjekket 11.03.2021).

— (nov. 2019a). *EclipseLink/FAQ/Security - Eclipsepedia*. [Online; accessed 19. Apr. 2021]. URL: https://wiki.eclipse.org/EclipseLink/FAQ/Security.

— (mai 2021a). *Jakarta Bean Validation specification*. [Online; accessed 13. May 2021]. URL: https://jakarta.ee/specifications/bean-validation/2.0/bean-validation_2.0.html (sjekket 13.05.2021).

— (mai 2021b). *Jakarta Concurrency*. [Online; accessed 13. May 2021]. URL: https://jakarta.ee/specifications/concurrency/2.0/concurrency-spec-2.0.html (sjekket 13.05.2021).

— (mai 2021c). *Jakarta Contexts and Dependency Injection*. [Online; accessed 13. May 2021]. URL: https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html (sjekket 13.05.2021).

— (mai 2021d). *Jakarta JSON Binding*. [Online; accessed 13. May 2021]. URL: https://jakarta.ee/specifications/jsonb/2.0/jakarta-jsonb-spec-2.0.html (sjekket 13.05.2021).

— (mai 2021e). *Jakarta RESTful Web Services*. [Online; accessed 13. May 2021]. URL: https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.html (sjekket 13.05.2021).

— (mai 2021f). *Jakarta Security*. [Online; accessed 13. May 2021]. URL: https://jakarta.ee/specifications/security/2.0/jakarta-security-spec-2.0.html (sjekket 13.05.2021).

— (apr. 2021g). *MicroProfile Runtimes Overview | The Eclipse Foundation*. [Online; accessed 23. Apr. 2021]. URL: https://www.eclipse.org/community/eclipse_newsletter/2019/november/2.php.

— (22. nov. 2019b). *MicroProfile/JWT Auth - Eclipsepedia*. [Online; accessed 13. May 2021]. URL: https://wiki.eclipse.org/MicroProfile/JWT_Auth.

Eclipse Process Framework (16. des. 2009). *Guideline: Entity-Control-Boundary Pattern*. [Online; accessed 14. May 2021]. URL: https://www.utm.mx/~caff/doc/OpenUPWeb/openup/guidances/guidelines/entity_control_boundary_pattern_C4047897.html.

Fielding, Roy T. og Julian Reschke (jun. 2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. DOI: 10.17487/RFC7231. URL: https://rfc-editor.org/rfc/rfc7231.txt.

Fielding, Roy Thomas (aug. 2000). *Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)*. [Online; accessed 28. Apr. 2021]. URL: https://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm#fig_5_8.

Flutter (software) (feb. 2021). en. Page Version ID: 1008450400. URL: https://en.wikipedia.org/w/index.php?title=Flutter_(software)&oldid=1008450400 (sjekket 03.03.2021).

Fowler, Martin og James Lewis (13. mai 2015). *Microservices*. [Online; accessed 18. May 2021]. URL: https://martinfowler.com/articles/microservices.html.

Git homepage (feb. 2021). URL: https://git-scm.com/ (sjekket 25.02.2021).

Github Pull request docs (feb. 2021). URL: https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests (sjekket 25.02.2021).

Globalization og Localization Association (2021). *Language Services*. GALA Global. URL: https://www.gala-global.org/knowledge-center/about-the-industry/language-services (sjekket 22.04.2021).

Goncalves, Antonio (2009). *Beginning Java EE 6 Platform with GlassFish 3 : from novice to professional*. Berkeley, CA New York: Apress Distributed to the Book trade worldwide by Springer-Verlag New York. ISBN: 9781430228905.

Google (mar. 2021). *Create and manage virtual devices*. en. URL: https://developer.android.com/studio/run/managing-avds (sjekket 08.03.2021).

Harrington, Jan L (2009). *Relational database design and implementation : clearly explained*. 3rd ed. Amsterdam ; Morgan Kaufmann/Elsevier. ISBN: 1-282-25841-9.

Harsh og Hemant (2019). *Building Microservices Applications on Microsoft Azure: Designing, Developing, Deploying, and Monitoring*. 1st ed. 2019. Berkeley, CA: Apress : Imprint: Apress. 271 s. ISBN: 978-1-4842-4828-7.

Hatzivasilis, George (2017). «Password-Hashing Status». I: *Cryptography* 1.2. Place: Basel Publisher: MDPI AG, s. 32. ISSN: 2410-387X.

Horizontal Pod Autoscaler (mar. 2021). [Online; accessed 13. May 2021]. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale (sjekket 13.05.2021).

IBM (mar. 2021a). *IBM Docs*. [Online; accessed 24. Apr. 2021]. URL: https://www.ibm.com/docs/en/was-liberty/base?topic=api-microprofile-config.

— (apr. 2021b). *IBM Docs*. [Online; accessed 11. May 2021]. URL: https://www.ibm.com/docs/en/was-liberty/base?topic=djr2al-configuring-resource-receive-multipartform-data-parts-from-html-form-submission-in-jax-rs-20 (sjekket 11.05.2021).

— (feb. 2019). *Java EE, Jakarta EE, MicroProfile, or maybe all of them*. [Online; accessed 23. Apr. 2021]. URL: https://developer.ibm.com/articles/java-ee-jakarta-ee-microprofile-or-maybe-all-of-them.

— (sep. 2017). *Open Liberty*. [Online; accessed 11. May 2021]. URL: https://www.openliberty.io/blog/2017/09/19/open-sourcing-liberty.html.

— (mai 2021c). *Open Liberty*. [Online; accessed 11. May 2021]. URL: https://openliberty.io/guides (sjekket 11.05.2021).

— (7. mai 2021d). *Open Liberty Docs*. [Online; accessed 13. May 2021]. URL: https://openliberty.io/docs/21.0.0.4/reference/feature/feature-overview.html.

Inc, GitHub (feb. 2021a). *Github homepage*. URL: https://github.com/ (sjekket 25.02.2021).

Inc., Docker (mai 2021b). *What is a Container? | App Containerization | Docker*. URL: https://www.docker.com/resources/what-container (sjekket 11.05.2021).

Ingress (21. apr. 2021). [Online; accessed 13. May 2021]. URL: https://kubernetes.io/docs/concepts/services-networking/ingress.

Internet Security Research Group (13. mai 2021). *About Let's Encrypt - Let's Encrypt*. [Online; accessed 13. May 2021]. URL: https://letsencrypt.org/about.

Jones, Michael, John Bradley og Nat Sakimura (mai 2015). *JSON Web Token (JWT)*. DOI: 10.17487/RFC7519. URL: https://rfc-editor.org/rfc/rfc7519.txt (sjekket 13.05.2021).

Juneau, Josh (jun. 2020). «Get started with concurrency in Jakarta EE». I: *Java Magazine*. URL: https://blogs.oracle.com/javamagazine/get-started-with-concurrency-in-jakarta-ee (sjekket 13.05.2021).

Kharod, Seema, Nidhi Sharma og Alok Sharma (sep. 2015). «An improved hashing based password security scheme using salting and differential masking». I: *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*. 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), s. 1–5. DOI: 10.1109/ICRITO.2015.7359225.

Kong Inc (mar. 2021). *Insomnia Desktop API Design Editor and API Client*. URL: https://insomnia.rest/products/insomnia (sjekket 08.03.2021).

Kurose, James F og Keith W Ross (2017). *Computer networking : a top-down approach*. 7th edition, Global edition. Boston: Pearson. 852 s. ISBN: 978-1-292-15359-9.

Lacoste, Jerome (jun. 2020). *Exec Maven Plugin – Introduction*. [Online; accessed 17. May 2021]. URL: https://www.mojohaus.org/exec-maven-plugin (sjekket 17.05.2021).

Liu, Cricket (2006). *DNS and BIND*. Sebastopol, CA: O'Reilly. ISBN: 9780596100575.

M. Haekal og Eliyani (28. okt. 2016). «Token-based authentication using JSON Web Token on SIKASIR RESTful Web Service». I: *2016 International Conference on Informatics and Computing (ICIC)*. 2016 International Conference on Informatics and Computing (ICIC). Journal Abbreviation: 2016 International Conference on Informatics and Computing (ICIC), s. 175–179. DOI: 10.1109/IAC.2016.7905711.

Material Design (2021). en. URL: https://material.io/resources (sjekket 03.03.2021).

Microsoft (sep. 2020). *Comparing WSL 2 and WSL 1*. URL: https://docs.microsoft.com/en-us/windows/wsl/compare-versions (sjekket 12.04.2021).

— (2021a). *Observer Design Pattern*. URL: https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern (sjekket 22.04.2021).

— (jun. 2017). *Sidecar pattern - Cloud Design Patterns*. [Online; accessed 5. May 2021]. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar.

— (10. mai 2021b). *The TypeScript Handbook*. [Online; accessed 14. May 2021]. URL: https://www.typescriptlang.org/assets/typescript-handbook.pdf.

Modulecounts (mai 2021). [Online; accessed 8. May 2021]. URL: http://www.modulecounts.com (sjekket 08.05.2021).

Mozilla Developer Network (13. mai 2021). *JavaScript language resources - JavaScript | MDN*. [Online; accessed 14. May 2021]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources.

NORID (apr. 2021). *Om Norid – Norid*. [Online; accessed 27. Apr. 2021]. URL: https://www.norid.no/no/omnorid.

Norman, Donald (2013). *The design of everyday things*. New York, New York: Basic Books. ISBN: 9780465003945.

NTNU (2021). *IMT4309*. [Online; accessed 9. Apr. 2021]. URL: https://www.ntnu.edu/studies/courses/IMT4309/2016#tab=omEmnet.

— (2020). *Zoom videoundervisning - Wiki - innsida.ntnu.no*. nb-NO. URL: https://innsida.ntnu.no/wiki/-/wiki/Norsk/Zoom+videoundervisning (sjekket 25.02.2021).

Nätt, Tom Heine (13. mai 2021). *JavaScript – Store norske leksikon*. [Online; accessed 13. May 2021]. URL: https://snl.no/JavaScript.

Object–relational mapping (6. mai 2021). I: *Wikipedia*. Page Version ID: 1021719270. URL: https://en.wikipedia.org/w/index.php?title=Object%E2%80%93relational_mapping&oldid=1021719270 (sjekket 11.05.2021).

Oracle Corporation (jan. 2013). *Introduction to the Java Persistence API - The Java EE 6 Tutorial*. [Online; accessed 11. Mar. 2021]. URL: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

OWASP Top Ten Project (mar. 2021). *OWASP Top Ten Web Application Security Risks | OWASP*. [Online; accessed 19. Apr. 2021]. URL: https://owasp.org/www-project-top-ten.

Payment API - DIBS Technical Documentation (mar. 2021). [Online; accessed 13. May 2021]. URL: https://tech.dibspayment.com/easy/api/paymentapi (sjekket 13.05.2021).

Persistent Volumes (mai 2021). [Online; accessed 13. May 2021]. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes (sjekket 13.05.2021).

PostgreSQL (2021). *PostgreSQL: About*. [Online; accessed 17. Mar. 2021]. URL: https://www.postgresql.org/about (sjekket 17.03.2021).

React Router: Declarative Routing for React (nov. 2020). [Online; accessed 13. May 2021]. URL: https://reactrouter.com (sjekket 13.05.2021).

Red Hat (apr. 2021). *What is a REST API?* [Online; accessed 28. Apr. 2021]. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

Rescorla, E. (aug. 2018). *The Transport Layer Security (TLS) Protocol Version 1.3*. URL: https://www.rfc-editor.org/rfc/rfc8446.html (sjekket 11.05.2021).

Schwaber, Ken og Jeff Sutherland (nov. 2020a). *Scrum - Daily standup*. URL: https://www.scrumguides.org/scrum-guide.html#daily-scrum (sjekket 25.02.2021).

Schwaber, Ken og Jeff Sutherland (nov. 2020b). *Scrum - Scrum team*. URL: https://www.scrumguides.org/scrum-guide.html#scrum-team (sjekket 25.02.2021).

— (nov. 2020c). *Scrum Guide | Scrum Guides*. URL: https://scrumguides.org/scrum-guide.html#sprint-retrospective (sjekket 11.03.2021).

Secrets (feb. 2021). [Online; accessed 13. May 2021]. URL: https://kubernetes.io/docs/concepts/configuration/secret (sjekket 13.05.2021).

Sharp, Helen (2019). *Interaction design : beyond human-computer interaction*. Indianapolis, IN: Wiley. ISBN: 9781119547259.

Software documentation (11. apr. 2021). I: *Wikipedia*. Page Version ID: 1017269911. URL: https://en.wikipedia.org/w/index.php?title=Software_documentation&oldid=1017269911 (sjekket 06.05.2021).

Thakkar, Mohit (2020). «Introducing React.js». I: *Building React Apps with Server-Side Rendering: Use React, Redux, and Next to Build Full Server-Side Rendering Applications*. Red. av Mohit Thakkar. Berkeley, CA: Apress, s. 41–91. ISBN: 978-1-4842-5869-9. DOI: 10.1007/978-1-4842-5869-9_2. URL: https://doi.org/10.1007/978-1-4842-5869-9_2.

The Kubernetes Authors (mai 2021). *Kubernetes Documentation*. [Online; accessed 4. May 2021]. URL: https://kubernetes.io/docs/.

UiO (2021). *IN1050*. [Online; accessed 9. Apr. 2021]. URL: https://www.uio.no/studier/emner/matnat/ifi/IN1050/h20/pensum.html.

virtualisering – IT (22. aug. 2020). I: *Store norske leksikon*. URL: http://snl.no/virtualisering_-_IT (sjekket 11.05.2021).

Welcome to cert-manager (mai 2021). [Online; accessed 13. May 2021]. URL: https://cert-manager.io/docs (sjekket 13.05.2021).

Weld home page (apr. 2021). [Online; accessed 10. May 2021]. URL: https://weld.cdi-spec.org (sjekket 10.05.2021).

Wikipedia (jan. 2021a). *Adobe XD*. en. Page Version ID: 1002330437. URL: https://en.wikipedia.org/w/index.php?title=Adobe_XD&oldid=1002330437 (sjekket 03.03.2021).

— (mar. 2021b). *Android Studio*. en. URL: https://en.wikipedia.org/w/index.php?title=Android_Studio&oldid=1010382567 (sjekket 08.03.2021).

— (5. mar. 2021c). *Dart (programming language)*. I: *Wikipedia*. Page Version ID: 1010419144. URL: https://en.wikipedia.org/w/index.php?title=Dart_(programming_language)&oldid=1010419144 (sjekket 11.03.2021).

Wikipedia (feb. 2021d). *Discord (software)*. en. Page Version ID: 1008274409. URL: https://en.wikipedia.org/w/index.php?title=Discord_(software)&oldid=1008274409 (sjekket 25.02.2021).

— (4. mar. 2021e). *Docker (software)*. I: *Wikipedia*. Page Version ID: 1010239429. URL: https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1010239429 (sjekket 11.03.2021).

— (feb. 2021f). *IntelliJ IDEA*. en. URL: https://en.wikipedia.org/w/index.php?title=IntelliJ_IDEA&oldid=1005949546 (sjekket 08.03.2021).

— (4. mar. 2021g). *Java (programming language)*. I: *Wikipedia*. Page Version ID: 1010284158. URL: https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1010284158 (sjekket 11.03.2021).

— (feb. 2021h). *Jira (software)*. en. URL: https://en.wikipedia.org/w/index.php?title=Jira_(software)&oldid=1004279923 (sjekket 25.02.2021).

— (19. apr. 2021i). *Singleton pattern*. I: *Wikipedia*. Page Version ID: 1018643939. URL: https://en.wikipedia.org/w/index.php?title=Singleton_pattern&oldid=1018643939 (sjekket 22.04.2021).

— (13. mar. 2021j). *Unit testing*. I: *Wikipedia*. Page Version ID: 1011894568. URL: https://en.wikipedia.org/w/index.php?title=Unit_testing&oldid=1011894568 (sjekket 16.03.2021).

— (19. mar. 2021k). *Void safety*. I: *Wikipedia*. Page Version ID: 1013023852. URL: https://en.wikipedia.org/w/index.php?title=Void_safety&oldid=1013023852 (sjekket 22.04.2021).

— (18. mar. 2021l). *Windows Subsystem for Linux*. I: *Wikipedia*. Page Version ID: 1012847707. URL: https://en.wikipedia.org/w/index.php?title=Windows_Subsystem_for_Linux&oldid=1012847707 (sjekket 12.04.2021).

Wordfence (25. jun. 2018). *Understanding Password Authentication & Password Cracking*. Wordfence. URL: https://www.wordfence.com/learn/how-passwords-work-and-cracking-passwords/ (sjekket 26.04.2021).

Zoom Video Communications (feb. 2021). en. Page Version ID: 1008318368. URL: https://en.wikipedia.org/w/index.php?title=Zoom_Video_Communications&oldid=1008318368 (sjekket 25.02.2021).

Vedlegg

A Kildekode server

Se vedlakt mappe: `vedlegg/kildekode/server`

B Kildekode applikasjon

Se vedlakt mappe: `vedlegg/kildekode/mobile-app`

C Forprosjektrapport

Se vedlakt fil: `vedlegg/forprosjektrapport_bacheloroppgave.pdf`

D Kravspesifikasjon

Se vedlakt fil: `vedlegg/maoyi_kravspesifikasjon.pdf`

E Produktkrav

Se vedlakt fil: `vedlegg/produktkrav.pdf`

F Sprintrapporter fra Jira

Se vedlakt fil: `vedlegg/sprintrapporter.pdf`

G Sprint oppgaver fra Jira

Se vedlakt fil: `vedlegg/sprint_oppgaver.pdf`

H Retrospektivrapporter fra Jira

Se vedlakt fil: `vedlegg/sprint_retrospectives.pdf`

I GitHub-repository

https://github.com/Fish-app

Nils-Jarle, Erlend, Trygve

Màoyì Handelsplattform

NTNU
Norwegian University of
Science and Technology