Siri Borgen Sandnes
Ina Folland Hegg
Mikkel Stavelie

TraceGo

iOS-applikasjon for bestilling og sporing av varer

Bacheloroppgave i Dataingeniør
Veileder: Rituka Jaiswal
Mai 2024

Bacheloroppgave

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

NTNU
Kunnskap for en bedre verden

Siri Borgen Sandnes
Ina Folland Hegg
Mikkel Stavelie

TraceGo

iOS-applikasjon for bestilling og sporing av varer

Bacheloroppgave i Dataingeniør
Veileder: Rituka Jaiswal
Mai 2024

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

Kunnskap for en bedre verden

Abstrakt

Denne bacheloroppgaven er skrevet for oppdragsgiveren Solwr, en bedrift i Ålesund som leverer programvare og systemer til næringslivet. Oppgaven fokuserer på utviklingen av en iOS-mobilapplikasjon designet for aktører i dagligvarebransjen for å forbedre effektivitet i bestillings- og leveringsprosesser. Applikasjonen gir brukerne muligheten til å bestille varer fra en leverandør og spore leveranser i sanntid. På grunn av GDPR-restriksjoner kunne ikke Solwrs eksisterende API-er og databaser benyttes, så det ble opprettet en mock-database for utvikling og testing. Baksystemet er utviklet som en monolittisk systemarkitektur og administrerer databasen med informasjon om brukere, produkter, bestillinger og leveringsdetaljer. Mobilapplikasjonen er utviklet for iOS og har et moderne, intuitivt brukergrensesnitt, noe som var et krav fra oppdragsgiver. Skytjenesten Google Cloud Platform (GCP) benyttes for hosting av serveren som håndterer applikasjonserveren. Dette sikrer skalerbarhet og pålitelighet, samt støtte for kontinuerlig leveranse og integrasjon ved bruk av verktøy som Terraform og GitHub Actions for å automatisere oppdateringer og vedlikehold.

Gjennom hele utviklingsprosessen har teamet anvendt agile metoder og verktøy som Jira og Confluence for å sikre en strukturert og effektiv prosjektgjennomføring. Applikasjonen er bygget med Swift for frontend og Java for backend.

Resultatet ble en brukervennlig og effektiv applikasjon som forenkler logistikkprosessene for dagligvarebutikker, restauranter og kiosker ved bestilling av varer fra leverandører. Prosjektet viser hvordan moderne teknologier kan brukes til å skape innovative løsninger som tilfredsstiller brukernes behov.

Abstract

This bachelor's thesis is written for the client Solwr, a company in Ålesund that delivers software and systems to the business sector. The thesis focuses on the development of an iOS mobile application designed for actors in the grocery industry to improve efficiency in ordering and delivery processes. The application allows users to order goods from a wholesaler and track deliveries in real-time.

Due to GDPR restrictions, Solwr's existing API's and databases could not be used, so a mock-database was created for development and testing. The backend system is developed with a monolithic architecture and manages databases containing information about users, products, orders, and delivery details. The mobile application is developed for iOS and has a modern, intuitive user interface, which was a requirement from the client.

The Google Cloud Platform (GCP) cloud service is used for hosting the server that handles the application's backend. This ensures scalability and reliability, as well as support for continuous integration and deployment using tools like Terraform and GitHub Actions to automate updates and maintenance.

Throughout the development process, the team applied agile methods and tools such as Jira and Confluence to ensure a structured and efficient project execution. The application is built with Swift for the frontend and Java for the backend.

The result was a user-friendly and efficient application that simplifies the logistics processes for grocery stores, restaurants, and kiosks when ordering goods from wholesalers. The project demonstrates how modern technologies can be used to create innovative solutions that meet the users needs.

Forord

Vi vil gjerne takke:

- Våre kontaktpersoner hos Solwr, Liv Ersdal, Magnus Grande og Reinhard Dietzel for godt samarbeid og god hjelp under hele prosessen.
- Vår veileder Rituka Jaiswal for gode råd og veiledning gjennom prosjektperioden.

Innholdsfortegnelse

Abstrakt 1

Abstract 2

Forord 3

Liste av figurer 10

Liste av tabeller 12

Liste av kodeeksempler 13

Forkortelser 14

1 Introduksjon 1

1.1 Bakgrunn 1

1.2 Problemstilling og formål 1

1.2.1 Kravspesifikasjoner 1

2 Teori 2

2.1 Nettverksprotokoller 2

2.1.1 TCP/IP Modellen 2

2.1.2 Transmission Control Protocol 3

2.1.3 HTTP og HTTPS 3

2.1.3.1 Forespørsler 4

2.1.3.2 Responskoder 4

2.2 Skytjenester 5

2.2.1 Servere 5

2.2.1.1 Virtuell maskin 5

2.3 Sikkerhet 6

2.3.1 Autentisering og autorisasjon med JSON Web Tokens 6

2.3.2 BCrypt 8

2.4 Databaser . . . 8
2.5 Arkitektoniske mønster . . . 8
2.5.1 MVVM . . . 8
2.6 Designprinsipper . . . 9
2.7 DevOps . . . 9
2.7.1 Konteinere i Docker . . . 9
2.7.2 Kontinuerlig integrasjon/kontinuerlig leveranse . . . 10
2.8 Mikrotjenester/monolitt . . . 10
2.9 Programmeringsparadigmer . . . 13
2.9.1 Objekt-orientert programmering . . . 13
2.9.1.1 Coupling . . . 14
2.9.1.2 Cohesion . . . 14
2.9.2 Funksjonell programmering . . . 14
2.10 Versjonskontroll . . . 15
2.10.1 Branch . . . 15
3 Metode **16**
3.1 Planlegging . . . 16
3.2 Organisering i prosjektperioden . . . 16
3.2.1 Veileder . . . 17
3.2.2 Oppdragsgiver . . . 17
3.2.3 Gruppen . . . 17
3.2.3.1 Roller og arbeidsfordeling . . . 17
3.2.4 GitHub . . . 18
3.2.5 Kommunikasjon . . . 18
3.3 Arbeidsmetodikk . . . 19
3.3.1 Agile metoder og scrum . . . 19
3.3.1.1 Sprinter . . . 19
3.3.1.2 Jira og Confluence . . . 19
3.4 Teknologi . . . 20
3.4.1 Programmeringspråk . . . 20
3.4.1.1 Java . . . 20
3.4.1.2 Swift . . . 20

3.4.1.3 SQL 21
3.4.2 Biblioteker og rammeverk 21
3.4.2.1 Spring Boot 21
3.4.2.2 Spring Security 21
3.4.2.3 Project Lombok 22
3.4.2.4 SwiftUI 22
3.4.2.5 AVFoundation 22
3.4.2.6 Security 22
3.4.2.7 Core Data 22
3.4.3 Utviklingsverktøy 23
3.4.3.1 Xcode 23
3.4.3.2 IntelliJ 23
3.4.4 Figma 23
3.4.5 Google Cloud Platform 23
3.4.6 REST API 24

4 Resultater **25**
4.1 Den fullstendige løsningen 25
4.2 Design og diagrammer 25
4.2.1 Use-case diagram 25
4.2.2 Site Map 26
4.2.3 Systemarkitektur 27
4.2.4 Entitetsrelasjonsdiagram 27
4.2.5 Wireframes 28
4.3 Serverapplikasjon 31
4.3.1 Modell 31
4.3.2 Kontrollere 31
4.3.3 Repositories 32
4.3.4 Service 33
4.3.5 Konfigurasjon 33
4.3.6 Database 35
4.3.7 Sikkerhet 36
4.3.7.1 Brukerautentisering 36

4.3.7.2 Sikkerhetskonfigurasjon 37
4.3.7.3 Passordhashing med BCrypt 39
4.3.8 Infrastrucure as Code 41
4.3.8.1 Terraform 41
4.3.8.2 CI/CD Pipeline med Github Actions 42
4.3.8.3 Enhetstesting i pipelinen 42
4.3.8.4 Hemmeligheter og variabler 43
4.3.9 Testing og testmiljø 43
4.3.9.1 Rammeverk for testing 44
4.3.9.2 Intregrasjonstester 44
4.3.9.3 Repositorytester 45
4.3.9.4 Servicetester 45
4.3.9.5 Endepunktstesting med Postman 46
4.3.9.6 Belastningstest med Postman 46
4.3.10 REST API Dokumentasjon 47
4.3.11 Javadoc 47
4.4 Front-end 47
4.4.1 Brukergrensesnitt 48
4.4.1.1 Design 48
4.4.2 Funksjonalitet 48
4.4.2.1 Login 48
4.4.2.2 Opprett bruker 49
4.4.2.3 Navigasjon 50
4.4.2.4 Ordre historikk 51
4.4.2.5 Ordre 52
4.4.2.6 Produkt 52
4.4.2.7 Ny ordre 53
4.4.2.8 Innstillinger 55
4.4.2.9 Mørk modus 56
4.4.3 Sikkerhet 56
4.4.3.1 Keychain 56
4.4.4 REST API kommunikasjon 57

5 Diskusjon **60**
5.1 Prosessen . . . 60
5.1.1 Kommunikajson . . . 60
5.1.1.1 Oppdragsgiver . . . 60
5.1.2 Planlegging og tid . . . 60
5.1.2.1 Scrum . . . 61
5.2 Resultatet . . . 61
5.2.1 Mobilapplikasjonen . . . 61
5.2.1.1 Valg av programmeringsspråk . . . 61
5.2.1.2 Design . . . 62
5.2.1.3 Funksjonalitet . . . 62
5.2.1.4 Sikkerhet . . . 63
5.2.1.5 Integrasjon med Serverapplikasjonen . . . 63
5.2.2 Serverapplikasjon . . . 64
5.2.2.1 Valg av systemarkitektur . . . 64
5.2.2.2 Spring Cloud Gateway . . . 64
5.2.2.3 Implementasjon av Websockets . . . 64
5.2.2.4 Det potensielle fullstendige systemet . . . 65
5.3 Videre arbeid . . . 65
5.3.1 Push notifkasjoner . . . 65
5.3.2 Coredata Swift . . . 66
5.3.3 E-post passordgjenopprettelse . . . 66
5.3.4 HTTPS forbindelse med SSL/TSL . . . 66
5.3.5 Ferdigstilling og refaktorering av front-end . . . 66

6 Konklusjon **68**

7 Referanser **69**

Referanseliste **70**

A Vedlegg **74**
Appendix . . . 75
A Vedlegg - Forprosjektplan . . . 75

B Vedlegg - Prosjekthåndbok 75

C Vedlegg - KI Deklarasjon 75

D Vedlegg - Postman tester 75

E Vedlegg - Kildekode 75

Liste av figurer

2.1 TCP/IP Modellen. 2

2.2 Etablering av forbindelse mellom klient og server gjennom en three-way-handshake. 3

2.3 HTTP-forespørsel/respons syklus. 4

2.4 Eksempel på tre virtuelle maskiner som kjører i isolerte miljøer hos en vertsmaskin gjennom en hypervisor. 6

2.5 JSON Web Token struktur. 7

2.6 Eksempel på dekodet JWT streng. 8

2.7 Tre-lags systemarkitektur. 11

2.8 Eksempel på monolittisk systemarkitektur. 12

2.9 Eksempel på mikrotjeneste systemarkitektur. 13

2.10 Klasse og objekt. 14

3.1 Organisasjonskart. 16

3.2 Organisasjonsdiagrammet beskriver rollefordelingen i teamet. 18

4.1 Use case diagram. 26

4.2 Site map. 27

4.3 Monolittisk systemarkitektur. 27

4.4 Entitetsrelasjonsdiagram. 28

4.5 Login. 29

4.6 Bruker-registrering. 29

4.7 Landingsside. 29

4.8 Aktive ordre. 29

4.9 Tidligere ordre. 29

4.10 Opprette ordre. 30

4.11 Spring-rammeverkets konfigurasjons- og oppstartsprosesser. 34

4.12 Mange-til-en (M:1) kardinalitet. 35

4.13 JSON Web Token sekvensdiagram. 37

4.14 Flyten av en HTTP-forespørsel gjennom filterene. 38
4.15 Eksempel på hvordan tiden (i millisekunder) vokser eksponentielt med antall iterasjoner definert i kostfaktoren. 40
4.16 Eksempel på komponentene den fullstendige hashede strengen består av. 40
4.17 Innlogging. 49
4.18 Innlogging med feil tilbakemelding. 49
4.19 Innlogging når bruker ikke eksisterer. 49
4.20 Registrer bruker. 50
4.21 Registrer bruker: Tomme felter. 50
4.22 Registrer bruker: Manglende e-post. 50
4.23 Landingsside. 51
4.24 Ordre informasjon. 51
4.25 Produktinformasjon. 51
4.26 Aktive ordre. 52
4.27 Leverte ordre. 52
4.28 Filtrering av ordre. 52
4.29 Informasjon om produkt. 53
4.30 Søk etter produkt. 54
4.31 Filtrer etter produkttype. 54
4.32 Skann strekkode. 54
4.33 Plasser ordre. 54
4.34 Rediger og bekreft ordre. 54
4.35 Ordre plassert. 54
4.36 Innstillinger. 55
4.37 Innlogginsside i mørk modus. 56
4.38 Ordrehistorikk i mørk modus. 56
4.39 Filtrering i mørk modus. 56

Liste av tabeller

4.1 Oppsummering av Postman ytelsestester. 47

Liste av kodeeksempler

4.1 Eksempel på et eksponert endepunkt som kan brukes til å opprette nye ordre for alle autentiserte brukere. (OrderController.java) (Dokumentasjon av endepunktet er med hensikt utelatt da det er for langt til å inkludere). 32
4.2 Eksempel på en repositoryklasse med egendefinerte SQL spørringer. (CustomerRepository.java). 32
4.3 Eksempel på servicemetode som definerer en viktig del av forretningslogikken. (OrderService.java). 33
4.4 Eksempel på en bønne som defineres på metode-nivå for å injesere konfigurasjonen for Swagger inn i Spring-kontektsen. (SwaggerConfig.java). 34
4.5 SecurityFilterChainen som jobber sammen med JWTRequestFilter for å autentisere brukere for autoriserte endepunkt. (SecurityConfig.java). . . 38
4.6 JWTRequestFilter som blir kjørt som et egendefinert filter i SecurityFilterChainen. (JWTRequestFilter.java). 39
4.7 Definering av virtuell maskin. 41
4.8 Konfigurering av brannmur. 41
4.9 Eksempelet viser 'unit-tests' jobben som kjører enhetstestene i pipelinen. (terraform.yml). 42
4.10 Konfigurasjon for databasen som henter verdier fra GitHub Secrets. (application.properties). 43
4.11 Initialiserer spring-konteksten med 'application-test.properties' slik at testmiljøet bruker spesifikke konfigurasjoner for testing. 43
4.12 Integrasjonstest som tester endepunktet som lar en bruker logge inn. (CustomerControllerIntegrationTest.java). 44
4.13 Testing av SQL-spørringen 'findByCustomer(Customer customer) som returnerer en liste med alle ordre assosiert til en bruker. (CustomerRepositoryTest.java). 45
4.14 Testing av servicemetoden registerCustomer(Registrationbody registrationbody) i CustomerServiceTest (CustomerServiceTest.java). 45
4.15 Sjekk av kamera tillatelse. 55
4.16 Lagring av token ved bruk av Keychain. 57
4.17 Henting av ordre fra back-end. 58

Forkortelser

Alfabetisk liste over forkortelser:

- **API** Application Programming Interface
- **CI/CD** Continuos Integration/Continuos Delivery
- **CPU** Central Processing Unit
- **CRUD** Create, Read, Update and Delete
- **DevOps** Development and Operations
- **ERP** Enterprise Resource Planning
- **GCP** Google Cloud Platform
- **GDPR** General Data Protection Regulation
- **GTIN** Global Trade Item Number
- **HTTP** HyperText Transfer Protocol
- **HTTPS** HyperText Transfer Protocol Secure
- **IaC** Infrastructure as Code
- **IDE** Integrated Development Interface
- **iOS** iPhone Operating System
- **JSON** JavaScript Object Notation
- **JWT** JSON Web Token
- **MVVM** Model, View, View Model
- **NTNU** Norges Teknisk-Naturvitenskapelige Universitet
- **OOP** Objekt Orientert Programmering
- **PHP** Hypertext Preprocessor
- **SSL** Secure Sockets Layer
- **SQL** Structured Query Language
- **TLS** Transport Layer Security
- **UI** User Interface
- **VCS** Version Control System
- **VM** Virtuell Maskin
- **WCAG** Web Content Accessibility Guideline
- **WMS** Warehouse Management System

Del 1

Introduksjon

1.1 Bakgrunn

Solwr er en bedrift i Ålesund som blant annet leverer ulike programvarer og systemer til næringslivet, og en stor andel av deres kunder er dagligvareaktører. I dag tilbyr de systemet Trace, som er en ERP-plattform for handelslogistikk. Trace inneholder ulike moduler for ordreadministrasjon, lageroperasjoner og transporthåndtering [1].

De ønsker nå å utvikle en løsning som kan tilby kunder av ulike dagligvareleverandører en mobilapplikasjon som åpner opp for muligheten til å bestille varer og samtidig spore leveranser i sanntid.

1.2 Problemstilling og formål

Prosjektets mål er å utvikle en mobilapplikasjon som gir dagligvarebutikker, kiosker, restauranter og lignende industrier muligheten til å legge inn ordre for å bestille varer direkte fra leverandør, og i tillegg spore ordrene sine i sanntid. Applikasjonen skal ha et tydelig fokus på brukervennlighet og effektivitet, for å sikre en god handleopplevelse for brukere. Applikasjonen skal bidra til en enklere logistikkprosess og øke transparensen i leveringsprosessen, som igjen bidrar til en mer strømlinjeformet operasjon og bedre kundeopplevelse.

1.2.1 Kravspesifikasjoner

- **Intuitivt:** Brukergrensesnittet designes for iOS, og skal være moderne og intuitivt for brukere.
- **Backend:** Det skal utvikles et baksystem for prosjektet med en egen database.
- **Integrering:** Informasjon fra databasen skal være tilgjengelig i brukergrensesnittet. Dette innebærer informasjon om brukere, produkter og varebeholdning hos leverandøren, samt informasjon om leveringsstatus.

Del 2

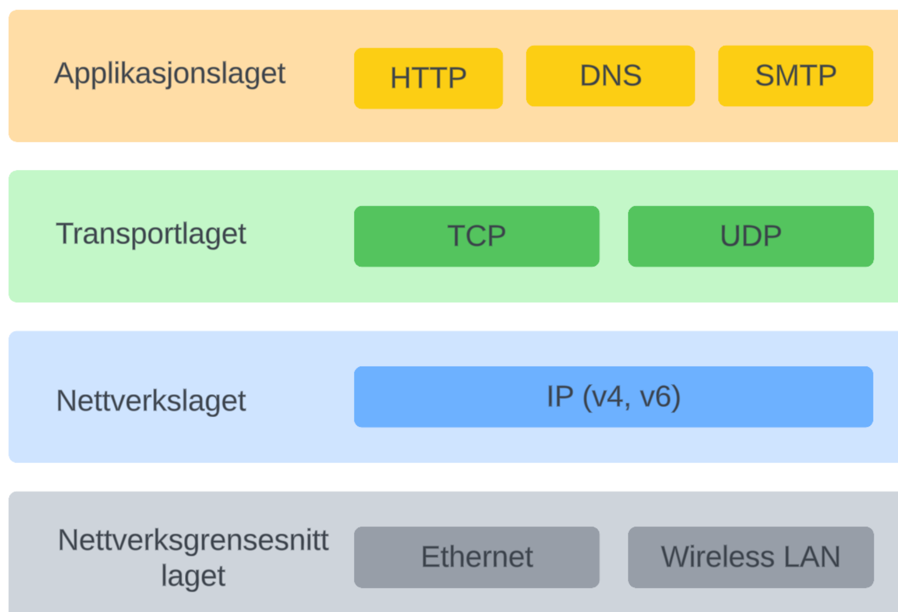
Teori

2.1 Nettverksprotokoller

Nettverksprotokoller kan beskrives som en rekke regler og standarder som gjør det mulig for enheter på et nettverk å kommunisere med hverandre. De ulike nettverksprotokollene definerer hvordan data kan overføres mellom enheter, dette inkluderer blant annet hvordan data pakkes, adresseres, sendes, mottas og tolkes. Nettverksprotokollene blir ofte beskrevet i OSI-modellen, eller TCP/IP modellen, som er en 'forkortet' versjon av OSI.

2.1.1 TCP/IP Modellen

TCP/IP (forkortelse for Transmission Control Protocol/Internet Protocol) er en gruppe nettverk- og kommunikasjonsprotokoller som benyttes for å koble sammen datamaskiner i nettverk på Internett [2]. Modellen består av fire lag: applikasjonslaget, transportlaget, nettverkslaget og nettverksgrensesnittlaget. Hvert lag har spesifikke funksjoner og protokoller som håndterer de forskjellige aspektene av datakommunikasjon. Figur 2.1 viser de ulike lagene i modellen og hvilke protokoller og funksjoner som inngår i hvert lag.

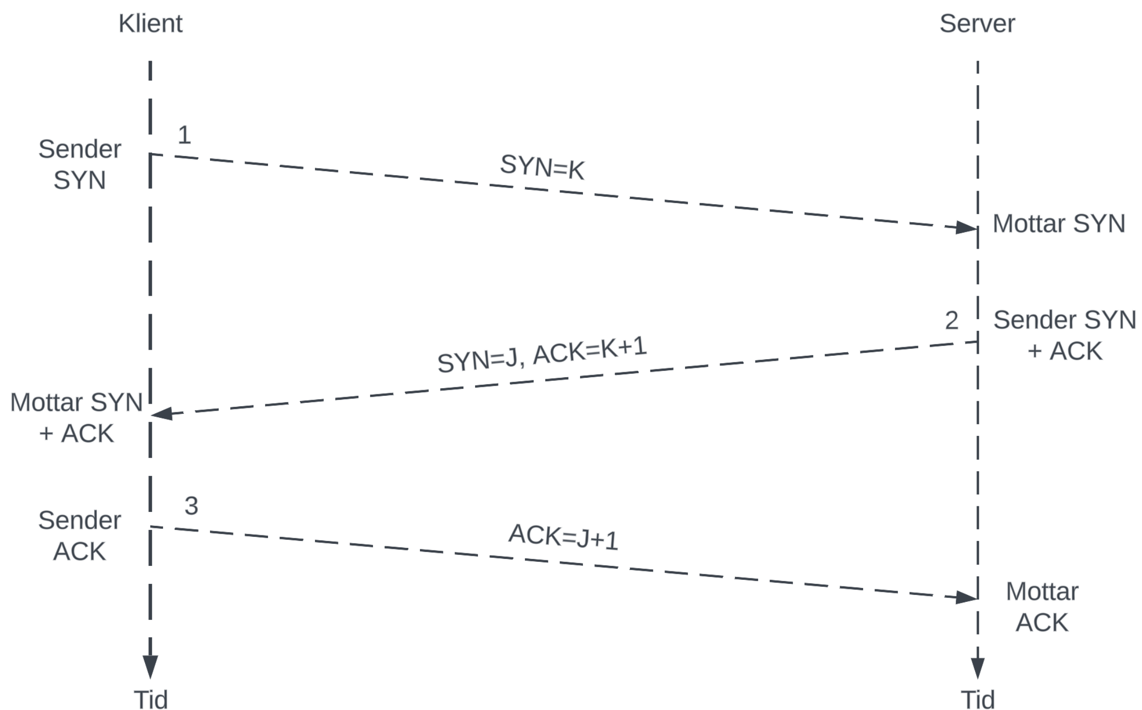


Figur 2.1: TCP/IP Modellen.

2.1.2 Transmission Control Protocol

TCP, eller Transmission Control Protocol, er en forbindelsesorientert protokoll som sikrer pålitelig overføring av informasjon mellom to parter [3]. Forbindelsen opprettes gjennom en 'three-way-handshake', som er essensiell for å opprette en pålitelig kommunikasjonskanal mellom klienten og serveren. Forbindelsen blir opprettet gjennom tre segmenter, hvor klienten først sender et SYN-segment til serveren for å initiere forbindelsen. Serveren svarer deretter med et SYN-ACK-segment, som både bekrefter mottakelsen av klientens SYN-flag og sender sitt eget SYN-flag for å opprette forbindelsen. Til slutt godtar klienten serverens SYN-flag ved å sende et ACK-segment, som bekrefter at forbindelsen er etablert [4].

Denne prosessen sikrer at både klienten og serveren er synkronisert og klare til å utveksle data på en pålitelig måte.



Figur 2.2: Etablering av forbindelse mellom klient og server gjennom en three-way-handshake.

2.1.3 HTTP og HTTPS

HTTP, eller Hypertext Transfer Protocol, er en protokoll som opererer på applikasjonslaget i TCP/IP-modellen og brukes til å overføre informasjon over internett mellom en klient og en server. Klienten, som kan være en nettleser eller en mobilbruker, sender en HTTP-forespørsel til serveren om tilgang til ressurser, og serveren svarer med de etterspurte ressursene dersom brukeren er autorisert. HTTP er dermed essensiell for kommunikasjonen og datautvekslingen som skjer på internett [5].

Figur 2.3: HTTP-forespørsel/respons syklus.

HTTPS, eller HyperText Transfer Protocol Secure, er også en protokoll som opptrer på applikasjonslaget i TCP/IP-modellen på samme måte som HTTP. Forskjellen mellom HTTPS og HTTP er at HTTPS bruker SSL/TLS (Secure Sockets Layer/Transport Layer Security) for å kryptere dataene som overføres mellom klienten og serveren. Når en HTTP-forespørsel blir sendt med HTTPS blir innholdet kryptert, og det er kun klienten og serveren som har tilgang til informasjonen som blir utvekslet [5].

2.1.3.1 Forespørsler

Når det samhandles med en server, for eksempel om handlinger skal utføres eller informasjon skal hentes, finnes det en rekke HTTP-forespørsler som benyttes for nettopp dette. Nedenfor er en oversikt over noen av de mest brukte forespørslene:

- **POST** er en forespørsel som sender data til serveren for å opprette en ressurs [6].
- **PUT** er en forespørsel brukt for å endre en ressurs på serveren [6].
- **GET** er en forespørsel som benyttes for å hente ressurser fra serveren [6].
- **DELETE** er en forspørsel som sletter en ressurs fra serveren [6].

2.1.3.2 Responskoder

Når serveren mottar en HTTP-forespørsel, returneres en responsskode til klienten. Denne koden gir informasjon om hvorvidt forespørselen var vellykket eller ikke. Det finnes en rekke kategoriserte responskoder som gir klienten informasjon om utfallet av forespørselen, enten det gjelder vellykkede utfall eller spesifikke feil som har oppstått. Nedenfor følger en oversikt over de ve vanligste kategoriene:

- **200-299** gir informasjon om vellykkede forespørsler [6].
- **300-399** gir informasjon om videresendinger eller omdirigeringer [6].
- **400-499** gir informasjon om feil fra klientsiden [6].
- **500-599** gir informasjon om feil fra serversiden [6].

2.2 Skytjenester

En skytjeneste kan betegnes som et bredt spekter av infrastruktur og tjenester som tilbys kunden over internett, via en ekstern leverandør. Det finnes ulike former for skytjenester, og det deles vanligvis opp i tre ulike modeller:

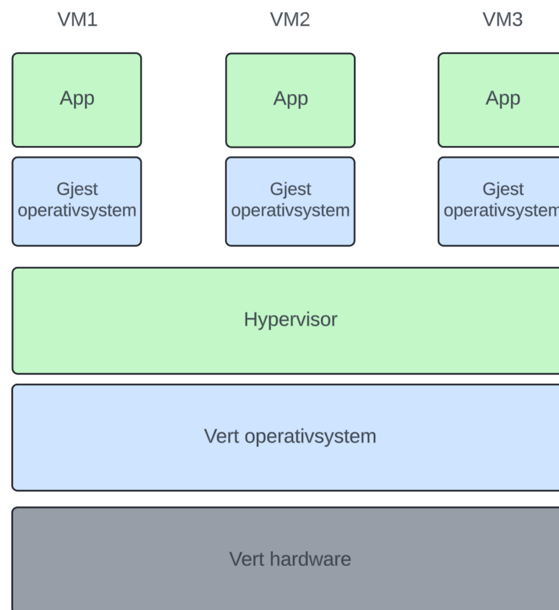
1. **IaaS, Infrastructure as a service:** Kunden kjøper tilgang til infrastruktur, som for eksempel servere, nettverk og virtuelle maskiner, og har frihet til å selv bestemme hvordan alt skal konfigureres. Ved bruk av IaaS slipper kunden å selv måtte investere i maskinvare [7].
2. **PaaS, Platform as a service:** Kunden kjøper tilgang til et skybasert miljø som inneholder alt som trengs for å kunne utvikle, kjøre, håndtere og utrulle applikasjoner og programvare, uten å selv måtte håndtere alt som ligger bak [8].
3. **SaaS, Software as a service:** Programvare, kunden kjøper tilgang til diverse programvarer og kan benytte seg av disse som de er [7].

2.2.1 Servere

En server er en datamaskin eller et program som tilbyr tjenester til andre datamaskiner, ofte kalt klienter. Tjenestene en server kan tilby inkluderer blant annet datalagringstjenester, som oppbevaring og administrasjon av data. En server kan også gi tilgang til ressurser, slik at data, filer og applikasjoner kan deles og brukes av klientene. I tillegg kan en server kjøre applikasjoner og håndtere programvare som klientene trenger. Det finnes altså mange forskjellige typer servere som tilbyr ulike tjenester, men sentralt for enhver server er behandlingen av forespørsler fra klienter. En server må være konfigurert til å lytte på forespørsler fra klientene. Når en server mottar en forespørsel fra en klient, svarer den med nødvendig respons, som sendes tilbake til klienten [9]. En illustrasjon som viser denne typen samhandling mellom en klient og en server kan sees i figur 2.3.

2.2.1.1 Virtuell maskin

En virtuell maskin, ofte forkortet til VM, er en programvarebasert emulering av en fysisk datamaskin [10]. Den utnytter deler av ressursene til vertsmaskinen, som CPU, minne, nettverk og lagring, for å kjøre et isolert operativsystem og applikasjoner gjennom virtualisering. Virtualiseringen foregår gjennom en hypervisor, som er programvare installert på vertsmaskinen som styrer gjestemaskinene. Hypervisoren utnytter deler av ressursene til vertsmaskinen, som CPU, minne, nettverk og lagring, for å kjøre isolerte operativsystemer og applikasjoner [11]. Dette tillater flere gjestemaskiner å kjøre samtidig på samme fysiske maskinvare, mens de opererer i separate, isolerte miljøer. Figur 2.4 viser tre virtuelle maskiner som kjører på en vertsmaskin gjennom en hypervisor.



Figur 2.4: Eksempel på tre virtuelle maskiner som kjører i isolerte miljøer hos en vertsmaskin gjennom en hypervisor.

2.3 Sikkerhet

I en moderne applikasjon er sikkerhet viktig, og det finnes mange ulike sikkerhetstiltak en kan implementere.

2.3.1 Autentisering og autorisasjon med JSON Web Tokens

JSON Web Tokens, også kjent som JWT's, er en åpen standard (RFC 7519) som brukes for å sikkert dele informasjon gjennom et JSON objekt i et kompakt, selvstendig format [12]. En egenskap som gjør JWT's til en effektiv og sikker løsning i moderne applikasjoner er at de signeres med en kryptografisk algoritme slik at informasjonen assosiert med en token ikke kan endres etter den er utstedt [13]. Dette betyr at mottakeren kan være sikker på at dataene er autentiske og brukeren er den den utgir seg for å være.

JSON Web Tokens har som oppgave å autorisere brukere ved å digitalt signere informasjon om brukerens identitet gjennom en kryptografisk algoritme. De vanligste algoritmene for digital signering er HS256 og RS256. HS256 er en symmetrisk algoritme som bruker samme nøkkel til både kryptering og dekryptering. RS256, derimot, er en asymmetrisk algoritme som benytter et nøkkelpar bestående av en offentlig og en privat nøkkel. Den offentlige nøkkelen kan deles fritt, mens den private nøkkelen må holdes hemmelig [14].

1. **Innlogging:** Klienten sender brukerinformasjon som e-post og passord til et eksponert endepunkt på serveren.
2. **Opprettelse av JWT:** Når brukeren er autorisert, genererer server en JWT for

klienten. JWT-en er en lang streng som inneholder autentiseringsinformasjon som identifiserer brukeren og gir tilgang til bestemte ressurser.

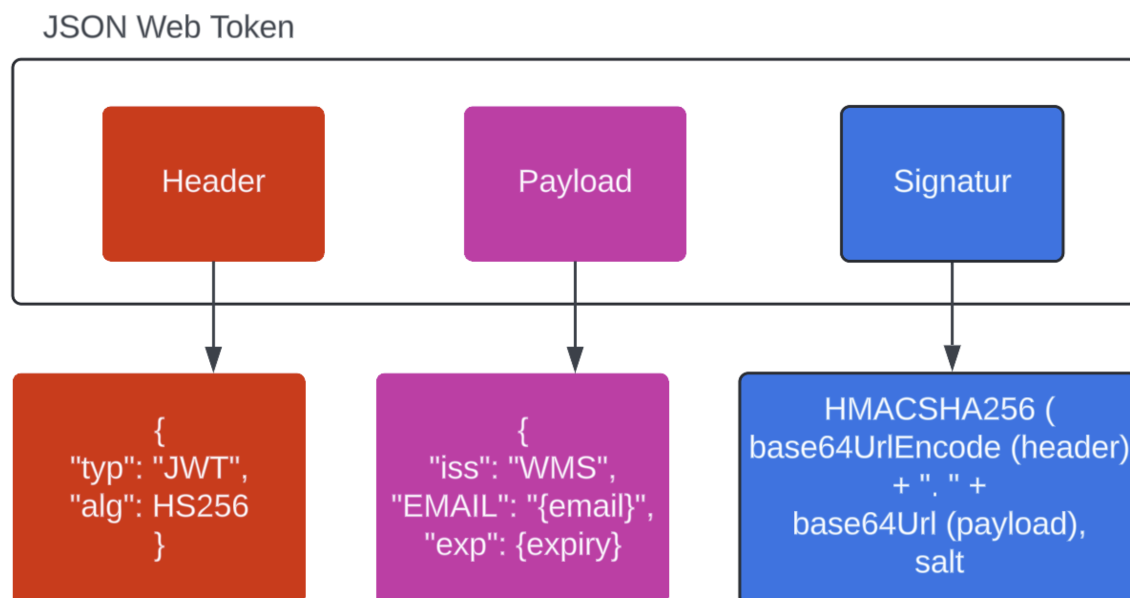
3. **Lagring av JWT:** Klienten lagrer JWT-et i sitt lokale minne for å bruke det i fremtidige forespørsler.

4. **Ressursforespørsel:** Klienten sender med JWT-et i headeren på HTTP-forespørselen for å få tilgang til beskyttede ressurser.

5. **Validering av JWT:** Når serveren mottar forespørselen, blir den validert og sammenlignet mot klienten for å sikre at den er gyldig.

6. **Svar på ressursforespørsel:** Dersom JWT-et er gyldig og brukeren er autorisert til de ressursene den etterspør, sendes disse med i HTTP-responsen.

Strukturen til en JSON Web Token består av tre ulike deler: en header, en payload og en signatur. Headeren består av informasjon som hvilken type token det er, og hvilken signeringsalgoritme som brukes. Payload-delen inneholder selve informasjonen, også kjent som 'claims', eller krav, som inkluderer identifkator om brukerens identitet, roller og annen relevant informasjon for autentisering. Dette gjør JWT's fleksible ettersom payloaden kan tilpasses applikasjonens sikkerhetskrav basert på hvor mye kontroll en bruker kan ha. Til slutt brukes signaturen til å bekrefte at avsenderen av tokenet er den den sier den er for å sikre at meldingen ikke ble endret underveis [15]. Dette gjør JWT's til en svært lettvektig og effektiv metode for autentisering og autorisering av brukere til å få tilgang til ressurser via API-et.



Figur 2.5: JSON Web Token struktur.

Når klienten etterspør ressurser fra API-et, sender den med JWT-et i headeren på HTTP-forespørselen som autorisasjon. Selvom JWT-et fra klientsiden bare består av en lang kodet streng, beholder den fortsatt strukturen med en header, payload og signatur atskilt med et punktum ('.'). Serveren dekoder og validerer strengen for å kontrollere brukerens identitet og tilgang. Figur 2.6 viser hvordan en dekodet JWT ser ut.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
.eyJpc3MiOiJXTVMiLCJFTUFJTCI6InN0YXZlbGllbUBnbWFpbC5jb20iLCJleHAiOjE3MTU1ODU5MDZ9
.YanAk7E-0aImSwbLWewpeuwq8tvRgS-qmaMil-Xb8dM

Figur 2.6: Eksempel på dekodet JWT streng.

2.3.2 BCrypt

Hashing av passord er et nødvendig sikkerhetstiltak for å beskytte brukernes sensitive informasjon mot uautorisert tilgang og potensielle datainnbrudd. BCrypt er et Java-bibliotek som implementeres gjennom Spring Security og tibyr hashing av passord gjennom kryptografiske algoritmer [16]. Det er nødvendig å benytte hashing istedenfor kryptering, da hashing er en enveis funksjon som er umulig å reversere til sammenligning med kryptering som kan reverseres [17].

2.4 Databaser

En database kan defineres som en samling, eller et oganisert sett av data. Det gjør det mulig å hente, endre, slette, lagre og administrere data. De fleste applikasjoner i dag er knyttet til en database hvor data er lagret, dette kan for eksempel være informasjon om brukere og brukernavn, varelager og utallige andre data som applikasjonen trenger tilgang til. En relasjonsdatabase kjennetegnes ved at data er lagret i tabeller, og forholdet mellom hver tabell er definert innad i databasen. I en nettbutikk kan for eksempel en bruker, og brukerens informasjon være lagret i en tabell, og ordre være lagret i en annen tabell. Disse tabellene er knyttet sammen med en primær- og fremmednøkkel, som gjør at brukeren blir knyttet til sine ordre [18].

2.5 Arkitektoniske mønster

Arkitektoniske mønster er typiske løsninger på vanlige problemer innenfor programvarearkitektur. Det finnes mange ulike mønstre som egner seg for forskjellige typer programmer. De arkitektoniske mønstrenes oppgave er å forenkle organiseringen og struktureringen av kode, for eksempel ved å definere hvordan man kan separere datahåndtering, brukergrensesnitt og interaksjonslogikk [19].

2.5.1 MVVM

MVVM er et populært arkitektonisk mønster, som blir mye brukt i programvareutvikling. Her er hovedmålet å separere brukergrensesnittet fra logikken i applikasjonen. De ulike komponentene applikasjonen blir delt inn i kalles Model, View og ViewModel [20].

1. Model: Inneholder all logikk og håndterer data som tilhører applikasjonen [20].
2. View: Inneholder brukergrensesnittet og det brukeren av appliksjonen ser [20].

3. ViewModel: Fungerer som et bindeledd mellom Model og View, når noe endres i Model komponenten, blir det sendt notifikasjoner gjennom ViewModel som gjør at View oppdateres [20].

2.6 Designprinsipper

Under utviklingen av systemer og applikasjoner spiller designprinsipper en viktig rolle. De bidrar til å gjøre applikasjoner brukervennlige og intuitive. En av de mest anerkjente tilnærmingene er Don Normans seks designprinsipper:

- Synlighetsprinsippet handler om at jo mer synlig et element er, jo mer sannsynlig er det at brukere benytter seg av det. Det samme kan sies andre veien, jo mindre synlig et element er, jo mindre sannsynlig er det at brukerne benytter seg av det.
- Prinsippet om tilgjengelighet handler om at funksjonaliteten til et element skal være tydelig, og at det ikke skal være noe tvil for brukeren om hvordan det skal brukes.
- Tilbakemeldingsprinsippet går ut på at det skal være klart for brukeren at en handling er utført, dette kan innebære taktile, visuelle og auditive responser på handlinger brukeren foretar.
- Begrensninger handler om at antall valg som er mulig å foreta for brukeren skal begrenses på en måte som gjør interaksjon med programmet intuitivt.
- Kartlegging handler om at det skal være en lettforståelig sammenheng mellom kontrollere og deres funksjon.
- Konsistensprinsippet handler om at elementer og funksjoner innad i et system skal være ensartede og følge de samme designmønstrene. Dette skal gjøre det enklere for brukerne å forstå hvordan de skal bruke systemet, kunnskap og erfaringer fra et område i systemet også kan benyttes i et annet.

Hvis alle disse designprinsippene benyttes under utviklingen av et system eller en applikasjon, skal de bidra til å sikre en god og intuitiv brukeropplevelse [21].

2.7 DevOps

2.7.1 Konteinere i Docker

En konteiner er en lettvektig enhet med programvare som pakker sammen all kode og dens avhengigheter slik at applikasjonen kan kjapt og pålitelig på hvert operativsystem uten behov for ytterligere konfigurasjon [22]. Dette gjør konteinere ideelle for kontinuerlig integrasjon og levering (CI/CD) da de sikrer konsistens på tvers av utvikling og produksjonsmiljøer.

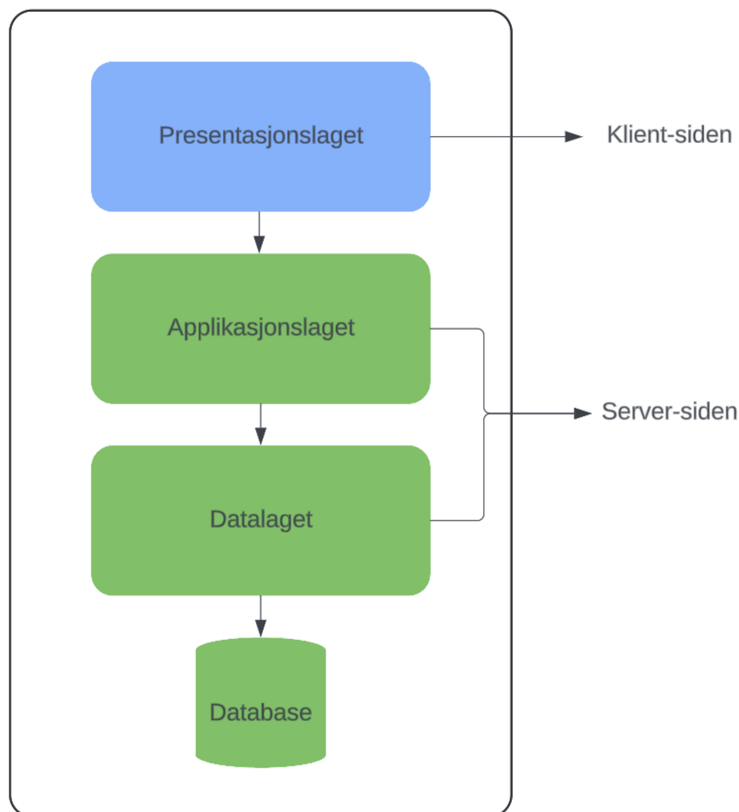
2.7.2 Kontinuerlig integrasjon/kontinuerlig leveranse

En kontinuerlig integrasjon/kontinuerlig leveranse pipeline, eller continous integration/continous delivery pipeline (CI/CD pipeline), gjør det mulig å automatisere prosessene som inngår i kontinuerlig integrasjon og kontinuerlig leveranse. Prosesser som bygging og testing skjer automatisk når endringer i kode blir lagt inn (committed), noe som bidrar til å sikre at feil oppdages tidlig og kvaliteten på koden opprettholdes. Dette resulterer i raskere og mer effektive prosesser innen programvareutvikling og utrulling. I tillegg kan automatisert utrulling redusere risiko for menneskelige feil og forbedre pålitelighet [23].

2.8 Mikrotjenester/monolitt

Mikrotjeneste- og monolittarkitekturer representerer to forskjellige tilnærminger til systemarkitektur innen informasjons- og kommunikasjonsteknologi. Mikrotjenestearkitekturen deler applikasjonen opp i flere selvstendige tjenester, hver med sitt eget ansvarsområde, som kommuniserer seg imellom via HTTP-baserte API-er. På den andre siden bygger monolittarkitekturen på en mer tradisjonell tilnærming, der hele applikasjonen er integrert i en enhetlig og samlet struktur som kjører som én samlet enhet [24].

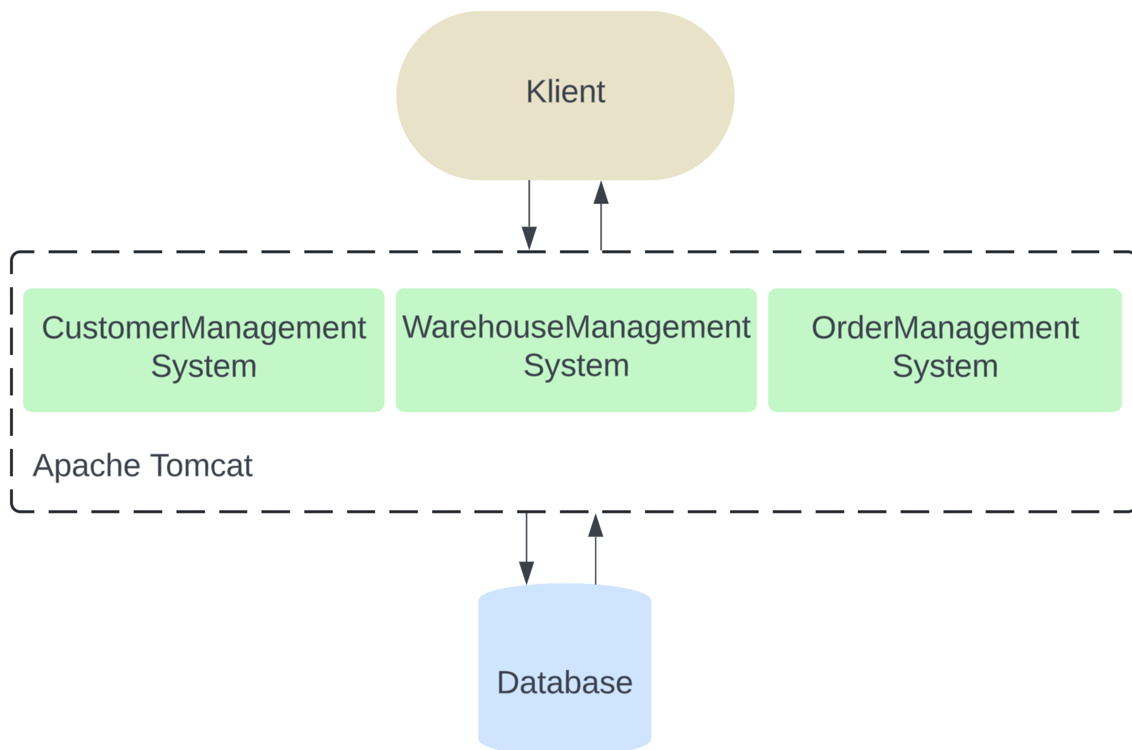
En monolittisk systemarkitektur fokuserer på å kjøre systemet som én applikasjon, bestående av flere komponenter med funksjonaliteter som er tett integrert [25]. Denne typen design legger vekt på enklehet i utvikling og drift, ettersom alle deler av systemer opererer som en samlet enhet. Komponenter som kjennetegner en slik tilnærming er som oftest en klient-side og en server-side tilkoblet en database. Klient-siden består som regel av HTML-sider i en nettleser, eller dersom klienten bruker en app på telefonen, vil den typisk være bygget med native teknologier eller kryssplattform-rammeverk. Server-siden håndterer applikasjonens baksystem-funksjoner som databasehåndtering operasjoner og sikkerhet. Den består vanligvis av en webserver eller en applikasjonsserver som jobber for å motta, prosessere og respondere til HTTP-forespørsler fra klient-siden. En slik struktur kan ofte refereres til en 'tre-lags arkitektur', bestående av et presentasjonslag, i dette tilfellet klient-siden, et applikasjonslag, som utgjør server-siden, og et datalag, som representerer databasen.



Figur 2.7: Tre-lags systemarkitektur.

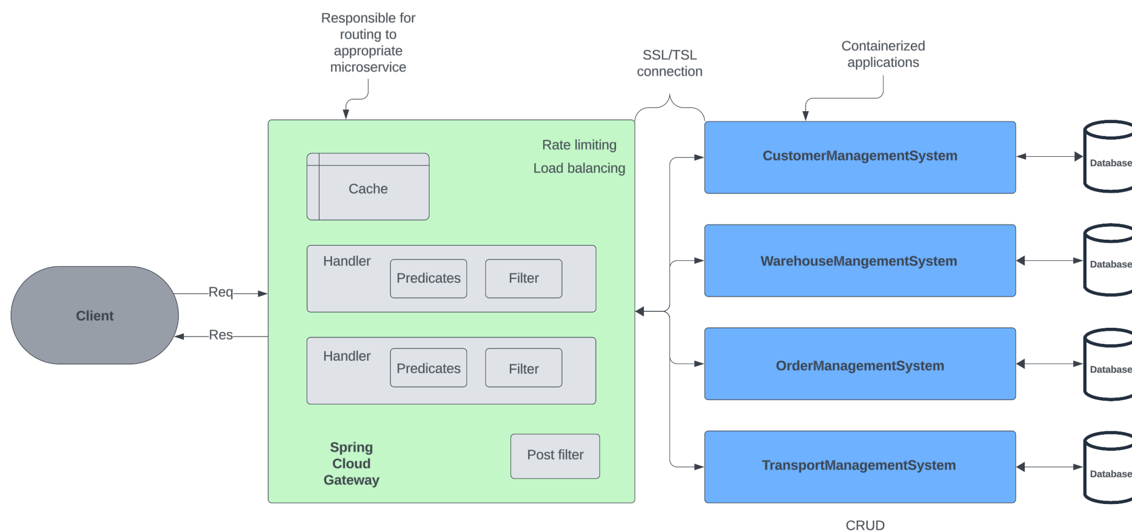
Monolittarkitektur gir flere fordeler for mindre organisasjoner eller bedrifter som ønsker høy hastighet til marked [26]. Fordeler med en slik arkitektur er simplisitet under utvikling da alle komponentene i applikasjonen er integrert i én kodebase. Dette gjør utviklingsprosessen hyppigere og mer strømlinjeformet. Når alle komponentene er samlet, blir kommunikasjonen mellom ulike deler av applikasjonen enklere, noe som kan føre til raskere feilsøking og testing [27]. Ytelse kan også være en fordel ved en monolittisk applikasjon, selvom dette avhenger av applikasjonens kompleksitet og datamengde. Som regel tilbyr monolittiske applikasjoner lav latens på klient-forespørsler siden forespørselen kun sendes gjennom få ledd uten noen form for ruting.

Selv om en monolittisk arkitektur tilbyr betydelige fordeler ved utvikling og enklere håndtering av prosesser, bærer den fortsatt preg av en rekke ulemper når applikasjonen skaleres og kompleksiteten økes. Vedlikehold av applikasjonen blir betydelig mer utfordrende siden alle komponentene er tett integrert og kompatible med hverandre. Dersom en eller flere komponenter feiler eller må erstattes, kan dette føre til feil i hele systemet og nødvendiggjøre omfattende testing [24]. Kontinuerlig leveranse blir også tregere og mer omfattende når hele systemet skal sendes gjennom 'pipelinen' hver gang det skal rulles ut en ny versjon. Figuren 2.8 viser hvordan et monolittisk system kan se ut.



Figur 2.8: Eksempel på monolittisk systemarkitektur.

På den andre siden har vi mikrotjeneste arkitektur som er en type arkitektur der deltjeneste som inngår i større systemer, kjører i separate konteinere og samarbeider via nettverk og API-er [28]. Dette betyr at systemet består av en samling med løst koplede komponenter som kjører uavhengig av hverandre i et lettvektig isolert miljø. En slik tilnærming fører til at komponentene er lett skalerbare, og tillater individuell utvikling og vedlikehold. Kontinuerlig leveranse gjennom CI/CD blir også mer hyppig og kostnadseffektiv da mikrotjeneste er lettvektige og inneholder kun de nødvendige pakkene for at tjenesten kan kjøres. Hver mikrotjeneste kan testes isolert, noe som effektiviserer testprosesser og minker tiden fra utvikling til produksjon. Dette reduserer risikoen for at én feil i en av komponentene påvirker resten av systemet, noe som styrker systemets robusthet og fleksibilitet. Figuren 2.9 viser hvordan en mikrotjeneste system kan se ut.



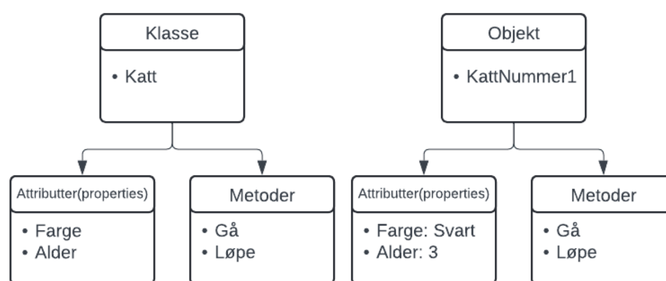
Figur 2.9: Eksempel på mikrotjeneste systemarkitektur.

Begge arkitekturene bygger på sine egenskaper og riktig arkitektur bør velges ut i fra prosjektets krav og teamets ekspertise og størrelse. En annen nøkkelfaktor ved å velge riktig arkitektur er å se prosjektets livsløp, og hvordan det kan utvikle seg og over tid. Dermed vil det også være nødvendig å tenke på skalerbarhet, videreutvikling og vedlikehold.

2.9 Programmeringsparadigmer

2.9.1 Objekt-orientert programmering

Objekt-orientert programmering er en programvareutviklingsmodell som organiseres rundt klasser og objekter, som er veldig nyttig når det kommer til å utvikle enklere og mer gjenbrukbar kode. Hvert objekt har egne attributter og egenskaper som defineres i klasser. En klasse fungerer som en oppskrift for individuelle objekter i et program. Den inneholder datafelter som representerer objektets tilstand, samt funksjoner og metoder som angir objektets atferd eller handlinger. En illustrasjon av dette kan sees i figur 2.10. OOP er en av de mest populære utviklingsmodellene når det kommer til utvikling av programvare, og det finnes mange ulike programmeringsspråk som bruker OOP [29].



Figur 2.10: Klasse og objekt.

Fundamentale prinsipper i objekt-orientert programmering inkluderer innkapsling, abstraksjon, arv og polymorfisme. Innkapsling beskytter objektets data og sikrer at tilstanden kun kan endres gjennom objektets egne metoder. Abstraksjon forenkler kompleksitet ved å skjule tekniske detaljer og eksponere kun nødvendige komponenter av objekter. Arv lar klasser overta egenskaper og metoder fra andre klasser, som fremmer gjenbruk av kode. Polymorfisme tillater at metoder kan brukes på forskjellige typer objekter, som gir økende fleksibilitet i kode. Samspillet mellom disse prinsippene gjør det mulig å utvikle mer modulær og gjenbrukbar kode, noe som gjør utvikling og vedlikehold av programvare enklere [29].

2.9.1.1 Coupling

Coupling, eller kobling, refererer til programvaremoduler, og hvor avhengig de er av hverandre. Det gjenspeiler hvor tett koblet forskjellige moduler i et programvaresystem er. Høy kobling (high coupling) forteller oss at moduler er tett sammenkoblet, og at eventuelle endringer i én modul, kan kreve endringer i en annen. Dette kan føre til utfordringer når det kommer til vedlikehold og endringer. Lav kobling (low coupling) kan beskrives som det motsatte, altså at ulike moduler ikke er tett sammenkoblet, og at endringer i en modul, enten påvirker en annen modul lite eller ikke i det hele tatt, noe som forenkler prosessen med vedlikehold og endringer [30].

2.9.1.2 Cohesion

Cohesion, eller kohesjon, referer til graden av tilhørighet og samstemthet mellom elementene innad i en modul, og hvorvidt de jobber sammen for å løse en bestemt oppgave. Høy kohesjon (high cohesion) vil si at elementene i modulen er samstemte og jobber mot samme oppgave, lav kohesjon (low cohesion) forteller oss at elementene ikke i høy grad er samstemte, og at de jobber mot flere forskjellige oppgaver. For å gjøre et system eller en programvare så enkel som mulig å vedlikeholde og videreutvikle, er det ønskelig at det skal være lav kobling og høy kohesjon [30].

2.9.2 Funksjonell programmering

Funksjonell programmering er en programvareutviklingsmodell som er inspirert av matematikken. I motsetning til objektorientert programmering, som organiseres rund

objekter og klasser, kjennetegnes funksjonell programmering av rene funksjoner og uforanderlighet. Rene funksjoner skal alltid returnere samme verdi til en gitt input, og skal være uten sideeffekter. Uforanderlighet refererer til at data ikke skal endres, som fører til at vi alltid vet at dataen er til stede, og at den er som du forventer [31].

Currying er et annet konsept i funksjonell programmering, som handler om å bryte ned en funksjon som tar flere argumenter til en kjede av funksjoner som tar ett argument av gangen. For eksempel, hvis vi har en funksjon som legger sammen to verdier, kan vi bruke currying for å lage en funksjon som først tar en verdi og deretter returnerer en ny funksjon som tar den andre verdien og deretter gir summen [32].

2.10 Versjonskontroll

Versjonskontroll med Git fungerer som en metode for å lagre endringer i en fil eller i en gruppe filer over tid. Dette systemet er nyttig fordi det tillater brukeren å sammenligne nyeste versjon med tidligere versjoner, og det gir også brukeren muligheten til å gjenopprette filene til en tidligere versjon, som kan være svært viktig om det dukker opp feil som brukeren ikke kan rette opp i. I et typisk versjonskontrollsystem lagres endringer i et 'repository'. Endringer lagres ved bruk av en handling kalt commit, hvor hver commit representerer en spesifikk versjon av filene med tilhørende notater (commit message) om hva som har blitt endret. Når man ønsker å se eller gjenopprette en tidligere versjon av filene, kan man bruke funksjonen 'checkout'. Dette gjør det mulig å hente ut og arbeide videre med enhver tidligere versjon som er lagret i repository [33].

2.10.1 Branch

En branch, eller gren, i Git kan benyttes for å gjøre det mulig å arbeide med ulike aspekter av et prosjekt parallelt uten å forstyrre hovedgrenen. Denne funksjonen lar utviklere opprette selvstendige grener som hver inneholder kopier av hovedgrenen. Branches benyttes ofte til å utvikle nye funksjoner, rette opp i bugs, eller å utvikle nye ideer i et isolert miljø. Dette reduserer risikoen for konflikter mellom pågående utviklingsarbeid og hovedgrenen, og gjør det mulig for team å utvikle endringer tryggere og mer effektivt. Når utviklingen på en branch er fullført og testet, er det mulig å integrere endringene tilbake til hovedgrenen, gjennom en prosess kalt merging. Dette sikrer at hovedgrenen forblir oppdatert med de siste godkjente endringene. Å ta i bruk branches kan altså være et nyttig verktøy når det kommer til samarbeid, og er viktig spesielt i større prosjekter med flere utviklere involvert [33].

Del 3

Metode

Dette kapittelet vil handle om hvordan prosjektet er utviklet. Dette innebærer både hvilke arbeidsmetoder som er benyttet, hvordan prosjektet er organisert, og hvilke programmeringsspråk og teknologier som er benyttet, og hvorfor.

3.1 Planlegging

I starten av prosjektperioden skulle gruppen levere en forprosjektplan, denne skulle hjelpe gruppen å bedre strukturere prosjektet. Forprosjektplanen inneholder en arbeidskontrakt, hvor det ble fordelt roller innad i gruppen, fastsatt prosedyrer for møteinnkallinger, varsling ved fravær, dokumenthåndtering og innlevering av arbeidskrav. Det ble også fastsatt en del retningslinjer i forhold til oppmøte, kommunikasjon og eventuelle uenigheter som skulle oppstå under prosjektet. I forprosjektplanen ble det også satt opp en oversikt over viktige datoer og milepæler under prosjektperioden, samt en mer detaljert oversikt over ansvarsområdene til det enkelte gruppemedlem. Forprosjektplanen er lagt til i denne rapporten som vedlegg.

Forprosjektplanen kan finnes som vedlegg under 'Forprosjektplan' i vedlegg A.

3.2 Organisering i prosjektperioden

Gjennom arbeidet med denne bacheloroppgaven har det hele tiden vært tre aktører involvert i prosessen. Organiseringen har bestått av 3 parter: oppdragsgiver, prosjektgruppe og veileder. Organiseringen vises i figur 3.1.

Figur 3.1: Organisasjonskart.

3.2.1 Veileder

I oppstarten av bachelorprosjektet fikk gruppen tildelt veileder fra NTNU, Rituka Jaiswal. Veileder skal bistå og hjelpe gruppen å sørge for at prosjektet følger en tilfredsstillende progresjon, og at nødvedig arbeid blir gjennomført. Veileder kan også bistå dersom det skulle oppstå utfordringer der bistand er nødvendig.

3.2.2 Oppdragsgiver

Oppdragsgiver for prosjektet er bedriften Solwr. Oppgaven som skulle løses, og alle kravspesifikasjoner tilknyttet prosjektarbeidet ble fremlagt av oppdragsgiver. Gruppen har gjennom prosessen hatt kommunikasjon med flere kontaktpersoner i selskapet, og i tillegg hatt tilgang til kontorplass sammen med dem.

Kommunikasjon med oppdragsgiver gjennom prosjektet har foregått både fysisk og digitalt. Det har blitt avholdt formelle møter med jevne mellomrom, der gjennomgang av progresjonen i prosjektet, og tilbakemelding fra arbeidsgiver var fokuset. I tillegg har det gjennom arbeid på samme kontor vært flere uformelle samtaler med bistand og råd for videre arbeid.

3.2.3 Gruppen

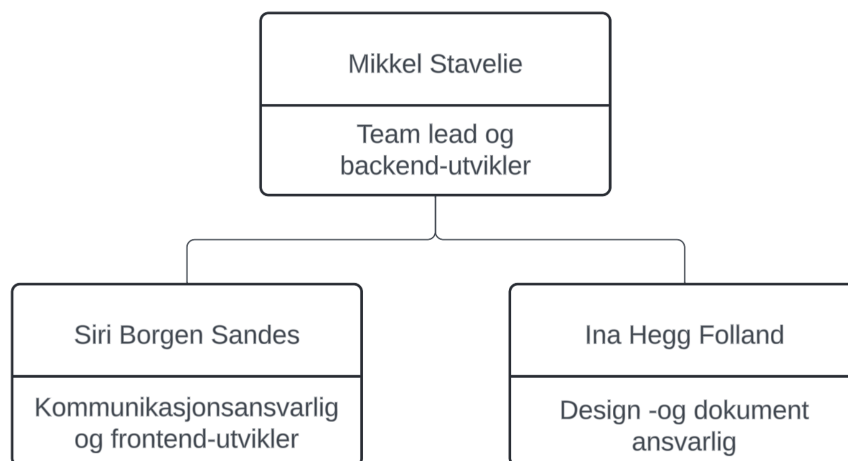
Bachelorgruppen består av tre dataingeniørstudenter, Siri Borgen Sandnes, Mikkel Stavelie og Ina Folland Hegg. Gruppen er ansvarlig for å utføre oppgaver gitt av oppdragsgiver på en måte som tilfredsstiller deres krav.

Gruppen skal også utarbeide en rapport som gir et korrekt og klart bilde på arbeidsprossessen gjennom perioden, teorier og teknologier som er benyttet, og resultatet av prosjektet.

3.2.3.1 Roller og arbeidsfordeling

Innad i gruppen ble ulike ansvarsområder og oppgaver fordelt basert på de enkelte gruppemedlemmenes styrker og interesser. Dette valget ble tatt på bakgrunn av konseptet 'cross functional teams', eller tverrfaglig team, som innebærer at utviklere med ulike fagområder eller ekspertiser settes sammen i et og samme team for å jobbe mot et felles mål. Dette kan medføre fordeler som økt effektivitet og raskere utvikling siden teamet jobber med ulike aspekter av prosjektet samtidig, og problemer og utfordringer kan adresseres raskere. Samtidig åpner det opp for ulike synspunkter til hver del av prosjektet fra forskjellige hold, som kan bidra til å øke innovasjon og nytenking [34].

Rollene ble definert allerede i oppstartsfasen, og la et grunnlag for forventningsstyring fra starten av.



Figur 3.2: Organisasjonsdiagrammet beskriver rollefordelingen i teamet.

3.2.4 GitHub

Gruppen valgte å bruke GitHub for å administrere all kode tilhørende prosjektet, dette er en nettbasert plattform som tilbyr versjonskontroll med Git, noe som kan være kritisk for å holde styr på endringer og fremgang i større prosjekter. GitHub er spesielt nyttig i teambaserte prosjekter, hvor det kan være nødvendig å spore hvem som har gjort hvilke endringer. GitHub kan bidra til å fremme samarbeid ved å tilby funksjoner som muliggjør at teammedlemmer kan arbeide uavhengig i forskjellige deler av prosjektet. Disse funksjonene sikrer at individuelle bidrag kan flettes sammen sømløst, uten å forårsake konflikter på grunn av endringer gjort av ulike gruppemedlemmer. Ved å benytte GitHub kan gruppen sikre at koden i prosjektet er godt organisert, lett tilgjengelig og beskyttet mot datatap [35].

3.2.5 Kommunikasjon

God kommunikasjon innad i gruppen, og mellom gruppen, veileder og oppdragsgiver er essensielt for å sikre god progresjon i prosjektet. Discord er et nettbasert kommunikasjonsverktøy som ble tatt i bruk tidlig i prosjektperioden, og all kommunikasjon som ikke ble gjort fysisk, ble gjort der. Discord tilbyr tjenester for både skriftlig kommunikasjon i form av meldinger og chat, samt tjenester for å avholde møter med lyd, video og mulighet for skjermdeling [36]. På denne måten kunne gruppe sikre god kommunikasjon uavhengig om alle hadde mulighet til å møtes for å jobbe på samme sted.

For kommunikasjon mellom gruppen og veileder ble epost flittig benyttet, sammen med både fysiske møter, og digitale møter via Zoom, som også tilbyr tjenester for video, lyd og skjermdeling.

Kommunikasjon og møter mellom gruppen og oppdragsgiver ble i hovedsak gjennomført fysisk, på kontoret hos oppdragsgiver. Et fåtall ganger under prosessen ble epost og meldingsfunksjonen på Discord også benyttet.

3.3 Arbeidsmetodikk

I planleggingsfasen ble det bestemt at agile metoder og scrum skulle benyttes under hele prosessen for å gjøre det så enkelt som mulig å ha oversikt over fremdrift og gjenstående arbeid.

3.3.1 Agile metoder og scrum

Agil metodikk kjennetegnes av smidighet og høy tilpasningsevne, hvor prosjektet brytes ned mindre deler. Dette fremmer kontinuerlig kommunikasjon mellom oppdragsgiver og utviklere, noe som forenkler planleggingen gjennom prosjektperioden. Gjennom denne tilnærmingen blir det enklere å integrere tilbakemeldinger og gjøre fortløpende forbedringer [37].

Scrum er et av rammeverkene under agil metodikk som blir hyppig brukt når det kommer til programvareutvikling. Alle issues, krav eller oppgaver legges inn i en backlog, en liste rangert etter hvor høyt den enkelte oppgaven skal prioriteres. Dette bidrar til at alle involverte ha oversikt over hvilket arbeid som er gjort, hva som må gjøres, og prioriteringsgraden til hver oppgave [38].

3.3.1.1 Sprinter

Fra start til slutt blir prosjektet delt opp i det som kalles sprinter. Hver sprint har normalt en varighet på en til fire uker. I begynnelsen av hver sprint avholdes et planleggingsmøte(sprint planning). Under dette møtet blir det bestemt hvilke oppgaver som skal fullføres i kommende sprint, disse oppgavene blir da flyttet fra backlog og inn i sprinten. Sprinten avsluttes med et møte hvor prosessen og arbeidet som ble gjort i sprinten evalueres. Dette deles inn i sprint review, hvor arbeidet evalueres, og sprint retrospective, hvor prosessen evalueres [38].

3.3.1.2 Jira og Confluence

Gruppen har brukt ulike verktøy for å holde styr på fremdrift, issues og dokumenter relatert til prosjektet. Programvarene Jira og Confluence ble tilbudt av NTNU.

Jira er en programvare som benyttes for å administrere sprinter, backlog og issues knyttet til et prosjekt, det gir også mulighet til å føre timelister, og gir en god oversikt over fremdrift og gjenstående arbeid. Storypoints benyttes for å estimere tidsbruk på hvert issue, og skal bidra for å gjøre planlegging mer oversiktlig [39].

I tillegg til Jira har gruppen benyttet programvaren Confluence. Begge programvarene er utviklet av Atlassian, og kan brukes sammen, slik at issues opprettet i Jira kan linkes til dokumenter og dokumentasjon i Confluence. Gruppen har brukt Confluence som en slags notatbok, hvor ulike diagrammer, møtereferater og andre notater lagres [39].

3.4 Teknologi

I dette prosjektet er det tatt i bruk mange ulike teknologier, dette delkapitlet vil ta for seg hvilke, og hvorfor de ble valgt.

3.4.1 Programmeringspråk

Ulike programmeringsspråk er tatt i bruk for utvikling av front-end og back-end i prosjektet, denne delen vil ta for seg hvilke, og hvorfor de ble valgt.

3.4.1.1 Java

Java er et objektorientert programmeringsspråk og er blant de mest brukte i verden. En stor pådriver til Javas popularitet er at progammer skrevet i Java kan kjøres uavhengig av plattform, noe som gjør det ideelt for blant annet backend-systemer som opererer på tvers av operativsystemer. Denne plattformuavhengigheten skyldes at Java-kode kompileres til en universell bytecode, som kan kjøres hvor som helst der det finnes en Java Virtual Machine (JVM). Java's slagord, 'Write once, Run anywhere', understreker denne evnen til å kjøre på tvers av plattform kort og konsist [40].

Gruppen valgte på bakgrunn av dette, kombinert med bred erfaring i språket å benytte Java i utviklingen av baksystemet.

3.4.1.2 Swift

I startfasen av prosjektperioden hadde gruppen møte med oppdragsgiver, hvor ulike teknologier og programmeringsspråk ble diskutert. Oppdragsgiver ønsket en mobil applikasjon for iOS, og hadde selv mye erfaring med bruk av Swift. Etter dette første møtet gjorde gruppen egen research om forskjellige programmeringsspråk, og det ble etter en stund avgjort at Swift skulle benyttes for frontend-delen av prosjektet. Ingen av gruppemedlemmene hadde tidligere erfaring med språket, og så derfor på dette som en unik læringsmulighet, siden oppdragsgiver var villig til å veilede og gi råd underveis.

Swift er Apples eget programmeringspråk for utvikling av applikasjoner til blant annet macOS og iOS. Språket ble introdusert i 2014, og har åpen kildekode. Siden introduksjonen har det vokst og blitt mer populært, og i dag blir en stadig økende andel apper for iOS utviklet med Swift [41]. Apple skriver selv at det var stort fokus på at språket skulle være enkelt å lære, selv for nybegynnere, og at det tilbyr enklere syntaks og derfor er enklere å forstå enn forgjengeren Objective-C. Et annet fokusområde for Swift var å øke sikkerheten og ytelsen sammenlignet med forgjengeren. Swift reduserer sjansen for programmeringsfeil som kan føre til at applikasjonen krasjer, takket være en strengere håndtering av nullverdier og en mer uttrykksfull syntaks for feilhåndtering [42].

3.4.1.3 SQL

SQL eller structured query language er et standardisert programmeringsspråk spesielt utviklet for å foreta spørringer. Det benyttes for å kunne hente ut, slette, legge til eller endre data i en database [43].

Spørringer, eller queries er instrukser som et databasesystem forstår. SQL-kommandoer kan deles inn i flere kategorier, basert på formålet de tjener:

1. DDL eller data definition language brukes for å opprette strukturen i databasen, og kan for eksempel brukes for å opprette tabeller. Noen eksempler på de mest brukte kommandoene i DDL er CREATE og DROP, som blant annet kan brukes for å opprette og slette en tabell [44].
2. DQL eller data query language benyttes fortrinnsvis for å hente ut data fra databasen. Her er en av de mest bruke kommandoene SELECT, som brukes for å hente ut spesifikk data [44].
3. DML eller data manipulation language brukes for å legge til ny data eller endre data i databasen, kommandoer som INSERT og UPDATE er hyppig brukt her for å legge inn eller endre data [44].
4. DCL eller data control language brukes av administratorer i databasen for å håndtere sikkerhet og adgangskontroll, her er kommandoer som GRANT og REVOKE brukt for å hendholdsvis gi og fjerne tilgang [44].
5. TCL eller transaction control language brukes for å håndtere transaksjonene i databasen, hvis det for eksempel er gjort endringer i databasen ved bruk av DML kommandoer, brukes TCL etterpå, og kommandoen COMMIT kan brukes for å lagre endringene, kommandoen ROLLBACK kan brukes for å angre endringene [44].

3.4.2 Biblioteker og rammeverk

3.4.2.1 Spring Boot

Spring Boot er en del av Spring Framework og tilbyr mange fordeler som hjelper utviklere med å bygge produksjonsklare applikasjoner raskt og effektivt [45]. Spring Boot inneholder en rekke komponenter og verktøy som gjør det til et av Java's mest populære rammeverk for å utvikle selvstendige Java-applikasjoner [46]. Siden Spring Boot applikasjonen lett kan pakkes inn i en selvstendig JAR-fil, betyr det at den kan den kjøres overalt som nevnt i avsnittet 3.4.1.1. Denne funksjonen gjør det lett å integrere applikasjonen i et 'Docker'-image, noe som videre forenkler distribusjonen, skalering og vedlikehold av applikasjonen.

3.4.2.2 Spring Security

Spring Security er et kraftig og svært tilpassbart rammeverk for autentisering og tilgangskontroll. Spring Security er på samme måte som Spring Boot, en del av Spring

Framework, og integrerer autentisering og autorisasjon for applikasjoner bygget med Spring Boot [47].

3.4.2.3 Project Lombok

Project Lombok, også kjent som bare Lombok, er et Java-bibliotek som fokuserer på å fjerne 'boilerplate'-kode ved bruk annotasjoner. Lombok tilbyr annotasjoner for å generere vanlige metoder som getters og setters (mutator- og accessor) metoder, toString, equals og hashCode, samt konstruktører både med og uten parametre [48]. En slik tilnærming kan bidra til bedre lesbarhet og redusere risikoen for feil. Lombok integreres også godt med Hibernate, som er en implementasjon av Java Persistence API (JPA), som er et annotasjonsbasert verktøy som brukes til å definerere databasestrukturen.

3.4.2.4 SwiftUI

SwiftUI er et deklarativt rammeverk for Swift, utviklet av Apple. I motsetning til selve språket Swift, har ikke SwiftUI åpen kildekode, som vil si at kildekoden ikke er offentlig tilgjengelig for brukere og utviklere å se, endre eller distribuere. SwiftUI ble først introdusert i 2019, og er et relativt nytt rammeverk[49]. Etter å ha vurdert de ulike alternativene for å utvikle en mobilapplikasjon med Swift, sto valget mellom UIKit og SwiftUI. Gruppen bestemte seg for å bruke SwiftUI, både for å sette seg inn i et relativt nytt rammeverk, og fordi oppdragsgiver mente at SwiftUI ville bli stadig mer populært og muligens etter hvert erstatte UIKit.

3.4.2.5 AVFoundation

AVFoundation er et rammeverk fra Apple som gjør det mulig å jobbe med multimedia i Objective-C og Swift. Dette omfatter avspilling, visning, innspilling og prosessering av både auditive og visuelle medier. Rammeverket gir utviklere muligheten til å bygge applikasjoner og programmer med ulike integrerte mediefunksjoner, for eksempel må AVFoundation benyttes for å kunne bruke kameraet på enheten applikasjonen kjører på [50].

3.4.2.6 Security

Security er et rammeverk fra Apple designet for å sikre data og kontrollere adgang til applikasjoner utviklet i Swift eller Objective-C. Rammeverket tilbyr en rekke verktøy og tjenester for å beskytte data, håndtere sertifikater og implementere kryptering, og bidrar dermed til å sikre at dataene er trygge og konfidensielle [51].

3.4.2.7 Core Data

Core Data er et rammeverk fra Apple som tilbyr løsninger for å håndtere objektene i modellaget i en applikasjon. Det tilbyr automatiserte løsninger for vanlige oppgaver knyttet til et objekts livssyklus, inkludert persistens ved bruk av spørringer [52].

3.4.3 Utviklingsverktøy

3.4.3.1 Xcode

Xcode er Apples eget IDE, eller integrerte utviklingsmiljø, som er gratis å bruke. Xcode kan brukes til å utvikle programvare til alle Apples operativsystemer blant anner iOS, macOS, tvOS og watchOS. I Xcode tilbys utviklere en rekke verktøy for å utvikle programvaredesign, programmere, feilsøke og teste, det støtter også kildekode fra en rekke ulike programmeringsspråk, blant annet Swift, Java, C++ og Objective-C [53]. Vi valgte å bruke Xcode siden applikasjonen vi utvikler skal være en iOS applikasjon, hvor kildekoden i brukergrensesnittet er Swift.

3.4.3.2 IntelliJ

IntelliJ er et IDE utviklet av selskapet Jetbrains, og er en av de mest populære utviklingsmiljøene for Java kildekode, men det støtter også en lang rekke andre programmeringsspråk. IntelliJ er kompatibel med en lang rekke plugins (utvidelser), og ulike byggeverktøy [54].

3.4.4 Figma

I det første møtet med oppdragsgiver kom det fram et felles ønske om å utarbeide detaljerte wireframes for alle sidene i applikasjonen. Dette skulle legge et solid grunnlag for den videre utviklingsprosessen og gjøre den mer strømlinjeformet, både med tanke på at gruppen hadde en visuell plan, og for å gjøre det enklere for oppdragsgiver å komme med tilbakemeldinger og ønsker angående design før det ble utviklet med kode. Gruppen hadde tidligere erfaring med bruk av Figma, og valgte derfor å fortsette med dette verktøyet også for dette prosjektet. Figma er kjent for sin effektivitet og brukervennlighet i designprosesser, noe som ville være til stor nytte i arbeidet med å skape detaljerte wireframes.

Figma er et digitalt verktøy som brukes for UI/UX design av applikasjoner. Det tillater flere brukere å samarbeide i sanntid, noe som gjør det til et godt verktøy for arbeid i team. I Figma kan utviklere og designere opprette wireframes som viser hvordan en applikasjon skal fungere, og hvordan den skal se ut, uten å skrive en eneste linje med kode. Dette bidrar til en raskere og mer effektiv prosess, der endringer kan visualiseres hurtig og tilbakemeldinger og ønsker fra oppdragsgiver kan implementeres raskt. Verktøyet har blitt populært blant designere for sin brukervennlighet, kraftige funksjoner og fleksibilitet, noe som gjør det til et verdifullt verktøy i digitalt design [55].

3.4.5 Google Cloud Platform

GCP, eller Google Cloud Platform, er en omfattende plattform som tilbyr en rekke ulike skytjenester for både utvikling og drift av applikasjoner. Blant de tilgjengelige tjenestene er IaaS, PaaS og SaaS som beskrevet i avsnitt 2.2. Ved å benytte IaaS, har det

blitt mulig å bruke IaC (Infrastructure as Code) verktøy for å definere infrastrukturen, inkludert virtuelle maskiner, nettverk og lagring, hos GCP.

3.4.6 REST API

En REST API (også kalt RESTful API) er et applikasjonsprogrammeringsgrensesnitt (API) som følger designprinsippene til 'representational state transfer' (REST) arkitekturstilen. REST APIer gir en fleksibel og lettvektig måte å integrere applikasjoner på og koble sammen komponenter på tvers av forskjellige systemer [56].

REST APIer kommuniserer gjennom HTTP-protokollen ved bruk av HTTP verb som beskrevet i 2.1.3.1, som definerer handlingen av HTTP-forespørselen. Informasjonen som utveksles mellom applikasjonene kan leveres i ulike formater som JSON, HTML, PHP eller klartekst. JSON (JavaScript Object Notation) er det mest populære formatet fordi det er lett leselig både for mennesker og maskiner, og det er programmeringsspråkagnostisk [56].

Del 4

Resultater

4.1 Den fullstendige løsningen

I denne delen vil det foretas en grundig gjennomgang av resultatene som ble oppnådd i prosjektet.

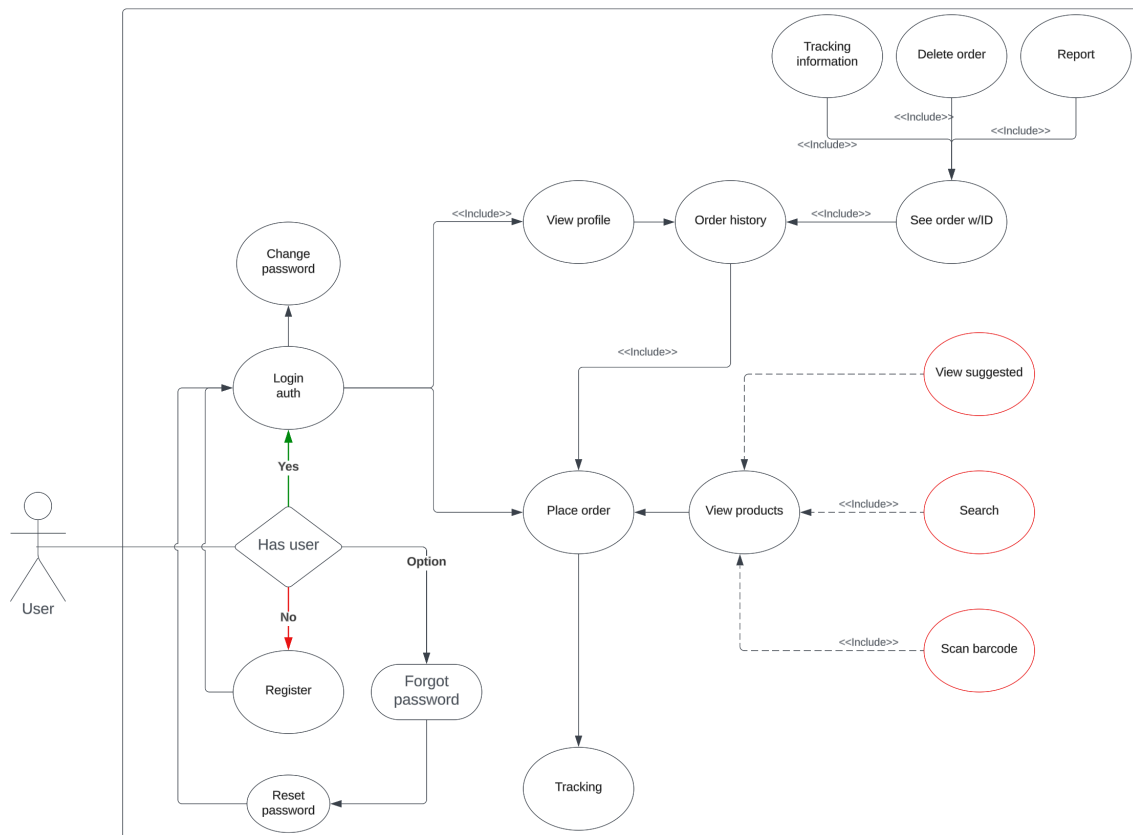
Fullstendig kildekode kan finnes som vedlegg under 'Kildekode' i vedlegg E.

4.2 Design og diagrammer

Design og diagrammer er essensielt for å kunne sette seg inn i og forstå hvordan et system fungerer, og få oversikt over prosessene og strukturen i systemet. Derfor har gruppen utviklet et utvalg diagrammer i løpet av prosessen som vil gjennomgås i denne delen.

4.2.1 Use-case diagram

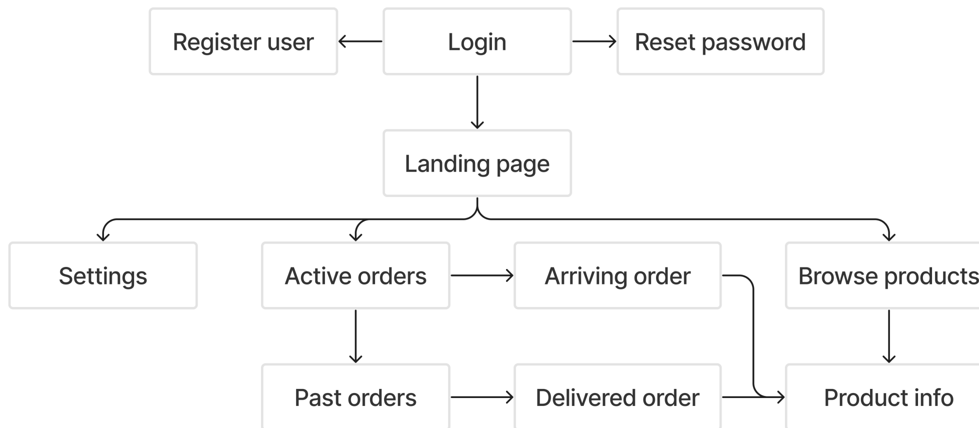
Use-case diagrammet gir en oversikt over hvilke valg brukeren kan gjøre under interaksjon med applikasjonen. Gruppen valgte å utvikle dette før utviklingen av selve applikasjonen for å visualisere hvordan brukerinteraksjonen skulle foregå. Å ha dette visualisert tidlig gav gruppen en tydelig utgangspunkt når det kom til utvikling av applikajsonen.



Figur 4.1: Use case diagram.

4.2.2 Site Map

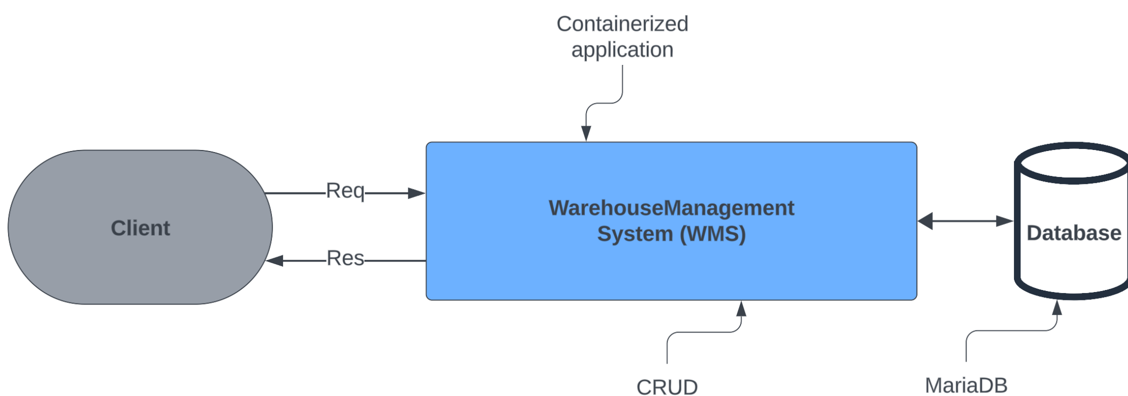
Etter anbefaling fra oppdragsgiver ble det utarbeidet et site map, dette kan sees i figur 4.2. Et site map er en oversikt over alle sider inne i applikasjonen, som også viser hvordan en kan navigere mellom sidene. Dette var et nyttig hjelpemiddel, og ble brukt under prosjektperioden for å kunne planlegge og visualiere hvordan navigasjonen mellom ulike sider i appen skal fungere.



Figur 4.2: Site map.

4.2.3 Systemarkitektur

Systemarkitekturen er en monolittisk arkitektur som bygger på 'tre-lags arkitekturen' beskrevet i 2.8. Arkitekturen beskriver hvordan komponentene i systemet kommuniserer sammen gjennom HTTP ved å gjennomgå HTTP-forespørsel/respons sykluser som vist i 2.3. Klienten initierer alltid forbindelsen til serveren og sender en HTTP-forespørsel som inneholder en bestemt handling eller forespørsel om data (req) som vist på figur 4.3. Baksystemet (WarehouseManagementSystem), som er en REST API, behandler forespørselen, kommuniserer med databasen, og sender tilbake en HTTP-respons (res) med etterspurte ressurser.

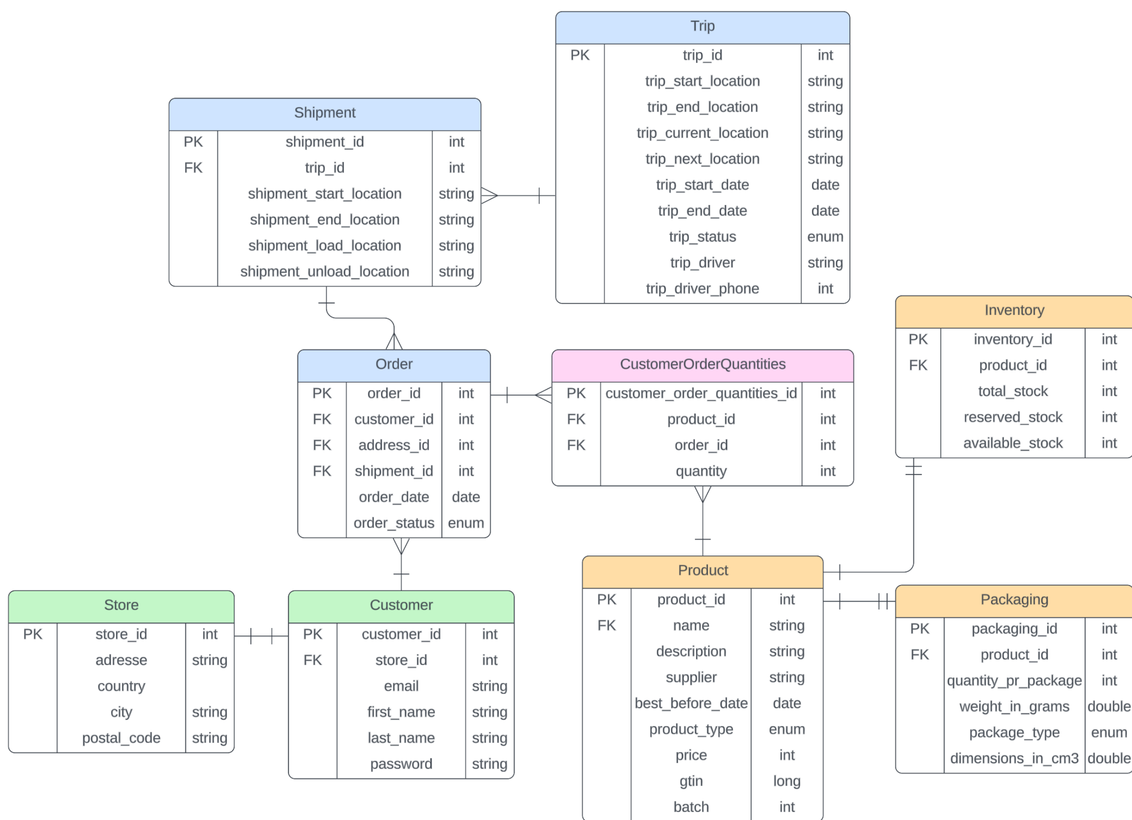


Figur 4.3: Monolittisk systemarkitektur.

4.2.4 Entitetsrelasjonsdiagram

Databasestrukturen er definert i et entitetsrelasjonsdiagram, ofte kalt E/R diagram (entity relation diagram), og viser alle tabellene, som er entiteter, og relasjonene mellom disse. E/R diagrammet gir en visuell fremstilling av hvordan data er organisert

i databasen og hvordan de ulike entitene forholder seg til hverandre gjennom ulike kardinaliteter. Kardinalitetene er tegnet som streker mellom tabellene og illustrerer forholdet mellom entitetene, for eksempel en-til-en(1:1), en-til-mange(1:M), eller mange-til-en(M:1). Figur 4.4 viser entitetsrelasjonsdiagrammet.

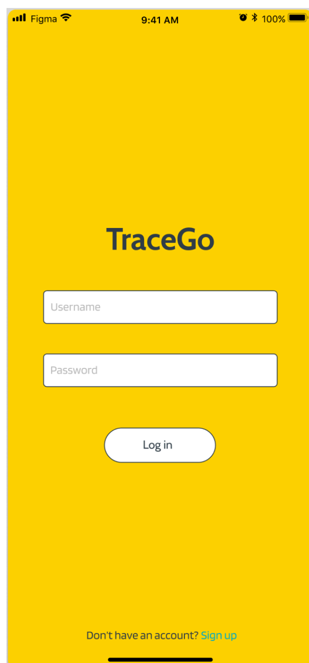


Figur 4.4: Entitetsrelasjonsdiagram.

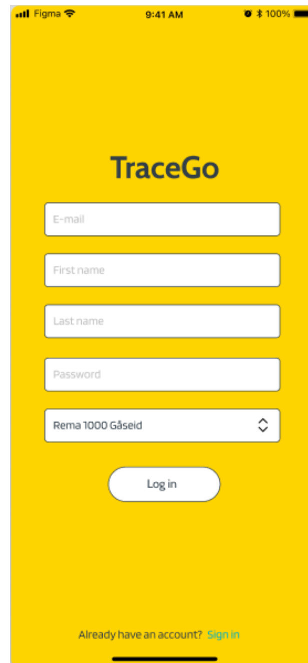
4.2.5 Wireframes

Designprosessen for applikasjonen ble startet i Figma, der gruppen utviklet et visuelt konsept basert på krav og en fargepalett levert av Solwr. Fargevalgene, som allerede var i bruk i Solwrs eksisterende systemer, bidro til å opprettholde kontinuitet i brukeropplevelsen og sikre at applikasjonens design var i tråd med bedriftens nåværende visuelle profil. Videre mottok gruppen spesifikasjoner om nødvendige funksjoner og sider som skulle inkluderes i applikasjonen, samt krav om at brukergrensesnittet skulle være intuitivt og enkelt å navigere. Utover disse forutsetningene, hadde gruppen frihet til å forme applikasjonens design etter egne ønsker og vurderinger.

Etter første utkast var ferdig, ble det avholdt et møte med oppdragsgiver, hvor design ble gjennomgått, og oppdragsgiver fikk mulighet til å komme med ønsker om endringer, og tilbakemelding på designet.

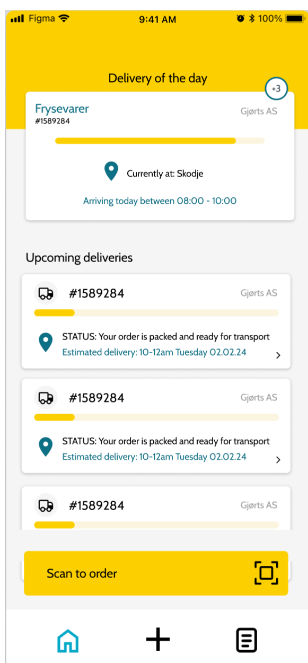


Figur 4.5: Login.

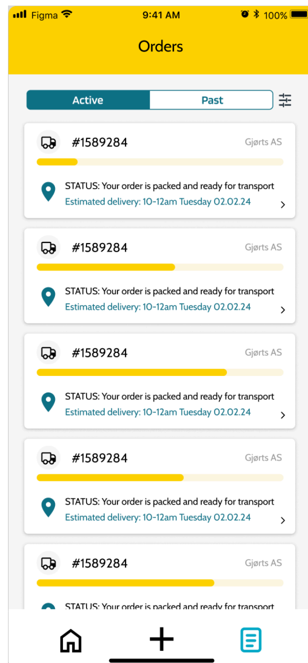


Figur 4.6: Bruker-registrering.

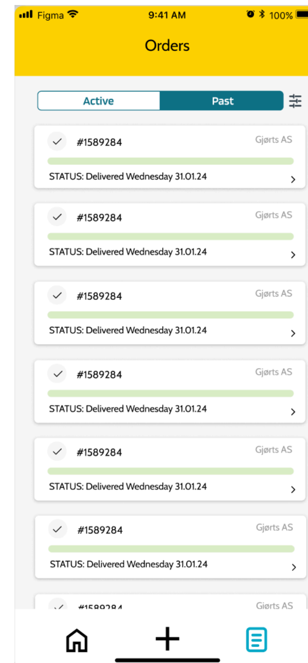
Under utviklingen av innlogging og bruker-registreringssidene, som vises i figur 4.5 og 4.6 var det stort fokus på at det skulle være intuitivt for brukeren, uten å måtte ha for mye informasjon på skjermen, derfor er det brukt plassholdere i tekstfeltene som forklarer brukeren hva som skal skrives inn. Hovedfargen på disse tre sidene er #FED400, en klar gulfarge som skaper blikkfang. Denne fargen er også gjenbrukt i de fleste sidene i applikasjonen som en aksent-farge.



Figur 4.7: Land-ingsside.

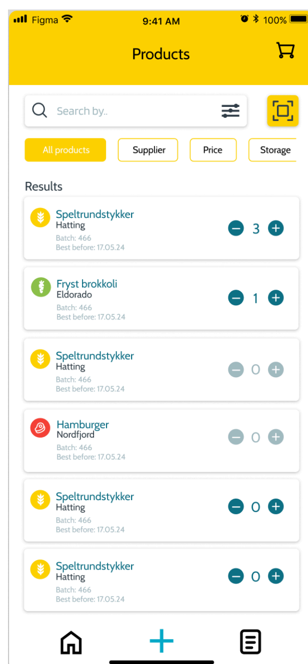


Figur 4.8: Aktive ordre.



Figur 4.9: Tidligere ordre.

Under utformingen av landingssiden i figur 4.7, prioriterte gruppen å fremheve informasjon om den neste leveransen som ankommer, som derfor opptar den største delen av siden, og er plassert øverst. Dette designvalget ble implementert for å sikre at brukeren umiddelbart kunne motta essensiell informasjon om forventet ankomsttid og nåværende lokasjon for leveransen. Videre på landingssiden ser brukeren flere aktive ordre, og status på disse. Til slutt ville gruppen at brukeren skulle kunne starte en ny bestilling fra landingssiden, og la derfor til en hurtigtast for vareskanning. Figur 4.8 og 4.9 viser sidene som tillater brukeren å veksle mellom visninger av aktive og tidligere ordre/leveranser. For å visualisere fremdriften i hver leveranse, implementerte gruppen en fremdriftslinje på både aktive og tidligere ordre. For aktive ordre ble aksentfargen benyttet, mens en grønn farge ble brukt for tidligere ordre for å indikere at leveransen er fullført og ordren er levert.



Figur 4.10: Opprette ordre.

Når brukeren blar gjennom produktlisten som vises i figur 4.10 og ønsker å legge til produkter i en ordre, har gruppen i designprosessen lagt vekt på å tilrettelegge for en enkel og intuitiv brukeropplevelse. Det er implementert en funksjon hvor den viktigste informasjonen om hvert produkt vises direkte i produktoversikten, uten at det er nødvendig å videre navigere inn på en spesifikt produkt. Informasjon som vises inkluderer produktnavn, leverandør, utløpsdato, og batchnummer. Dette tiltaket sikrer at brukeren raskt kan se essensielle detaljer om produktene.

For å ytterligere effektivisere brukerens søk etter spesifikke varer eller leverandører, er det øverst på siden lagt til et søkefelt. Dette skal gjøre det mulig å raskt finne frem til ønskede produkter.

4.3 Serverapplikasjon

Baksystemet er en REST API server utviklet med Java's Spring Boot rammeverk. Den er utviklet og strukturert via Spring Boots beste praktiser [57]. Som byggesystem har vi tatt i bruk Apache Maven som tillater automatisering av byggeprosesser og håndtering av avhengigheter på en effektiv måte i Java prosjekter [58]. Applikasjonen er distribuert på en virtuell maskin som kjører applikasjonen i en Docker konteiner, noe som gir en isolert og konsistent miljø for applikasjonen.

4.3.1 Modell

Modellklassene representerer selve skallet i applikasjonen og utgjør entitetene som blir manipulert og håndtert gjennom levetiden til systemet. De er fundamentale i hvordan dataene lagres og organiseres i databasen, og spiller en viktig rolle i applikasjonens datalag. De definerer strukturen og fungerer som et bindeledd mellom forretningslogikken og datalaget, og spesifiserer tabellene som opprettes i databasen via Hibernate, som er en implementasjon av Java Persistence API (JPA). Modellklassene tar i bruk Lombok-rammeverket for å redusere 'boilerplate'-kode ved å benytte annotasjoner som:

- **@NoArgsConstructor:** Genererer en konstruktør som ikke tar inn argumenter i konstruktøren.
- **@AllArgsConstructor:** Genererer en konstruktør som tar inn et argument for hvert felt i klassen.
- **@Getter:** Automatisk genererer accessor-metoder for alle felt i klassen.
- **@Setter:** Automatisk genererer mutator-metoder for alle felt i klassen.

Lombok og JPA fungerer godt sammen ettersom Lombok reduserer mengden 'boilerplate'-kode som kreves for å jobbe med JPA entiteter. Dette gjør utviklingen mer effektiv og mer lesbar.

4.3.2 Kontrollere

Kontrollerklassene er ansvarlige for å håndtere HTTP-forespørsler og manipulere HTTP-responser ved å tilpasse responskoden og innholdet til hver respons. De opptrer som et bindeledd mellom klienten og applikasjonsens forretningslogikk [59]. Kontrollerklassene avslører en rekke endepunkter som for eksempel '/api/orders' som er tilgjengelig for klienten for å nå eller manipulere de ressursene de etterspør. Metodene i kontrollerklassene er annotert med ulike typer mapping for å definere hva som forventes HTTP-forespørslene de skal håndtere. For eksempel brukes @GetMapping for å håndtere GET-forespørseler, mens @PostMapping brukes for å håndtere POST-forespørsler. Det blir også brukt @RequestBody for å tolke innkommende data i et JSON format, og @PathVariable for å definere variabler i URL-en som en del av HTTP-forespørselen. Kontrollermetodene returnerer et ResponseEntity<T> objekt som opptrer som en respons og kan manipuleres til å returnere data og HTTP-responskoder ut i fra hva som ønskes. Annotasjonen @AuthenticationPrincipal, som

er en del av Spring Security, blir også brukt for å injesere det autentiserte brukerobjektet direkte inn i en kontrollermetode for å hente eller manipulere data som er knyttet opp mot brukeren. Kode eksempel 4.1 viser et eksponert endepunkt som er tilgjengelig for autentiserte brukere.

Kode eksempel 4.1: Eksempel på et eksponert endepunkt som kan brukes til å opprette nye ordre for alle autentiserte brukere. (OrderController.java) (Dokumentasjon av endepunktet er med hensikt utelatt da det er for langt til å inkludere).

```
@PostMapping("/createorder")
    public ResponseEntity<?> createOrder(@AuthenticationPrincipal Customer
        customer, @RequestBody Order order) throws NotEnoughStockException {
        if (customer != null) {
            Order createdOrder = this.orderService.createOrder(order, customer)
                ;
            if (createdOrder != null) {
                return new ResponseEntity<>(createdOrder, HttpStatus.CREATED);
            } else {
                return new ResponseEntity<>("Order was not created", HttpStatus
                    .BAD_REQUEST);
            }
        }
        return new ResponseEntity<>("Customer is not authenticated", HttpStatus
            .FORBIDDEN);
    }
```

4.3.3 Repositories

Repositoryklassene brukes til å håndtere lagring og henting av data i en database. De tilbyr et abstraksjonslag mellom forretningslogikken og databasen, og utformer datalaget i en tre-lags arkitektur. Implementasjonen av et 'repository' arver fra CrudRepository<T, ID>, som igjen arver fra Repository<T, ID>. I vår implementasjon har vi benyttet ListCrudRepository<T, ID> som arver fra CrudRepository<> for å erstatte listeinnhentingen med List<E> i stedet for Iterable<E>, som CrudRepository<> tilbyr. Dette forenkler håndteringen av data i serviceklassene og gir en mer intuitiv tilgang til dataene. ListCrudRepository<> tilbyr direkte kontakt med databasen gjennom egendefinerte og predefinerte SQL spørringer, noe som gjør det enkelt å hente, sortere og lagre data i databasen. Repositoryklassene blir annotert med @Repository for å indikere at klassen er en del av datalaget og brukes til databaseoperasjoner. Kode eksempel 4.2 viser en repositoryklasse med egendefinerte SQL spørringer.

Kode eksempel 4.2: Eksempel på en repositoryklasse med egendefinerte SQL spørringer. (CustomerRepository.java).

```
@Repository
public interface CustomerRepository extends ListCrudRepository<Customer,
    Integer> {

    /**
     * Retrieves a Customer by their email address, ignoring case.
     *
     * @param email the email address of the Customer to retrieve.
     * @return an Optional containing the Customer, if found; otherwise, an
         empty Optional.
```

```
     */
    Optional<Customer> findByEmailIgnoreCase(String email);
}
```

4.3.4 Service

Serviceklassene inneholder den primære forretningslogikken og reglene i applikasjonen. De fungerer som et tjenestelag som forenkler kommunikasjonen mellom kontrollerklassene og repositoryklassene, og sikrer at datatilgangene er konsistente og at database-transaksjoner håndteres korrekt. Alle serviceklasser blir annotert med @Service for å vise at klassen inneholder forretningslogikk og bør behandles som et tjenestelag. Servicemetoder som utfører databasespørringer blir annotert med @Transactional for å sikre at databasetransaksjoner håndteres korrekt, effektivt og på en sikker måte gjennom at hele operasjonen blir fullført suksessfult, eller rulles helt tilbake ved en feil. Dette kjennetegner 'atomicity' i ACID-prinsippene (atomicity, consistency, isolation, durability) som beskriver de grunnleggende egenskapene i en transaksjonshåndtering i databasesystemer [60]. Kode eksempel 4.3 viser hvordan en viktig del av forretningslogikken fungerer. Siden flere brukere kan bestille varer til ulike butikker med forskjellige ønskede leveringsdatoer, må disse sorteres på en effektiv måte.

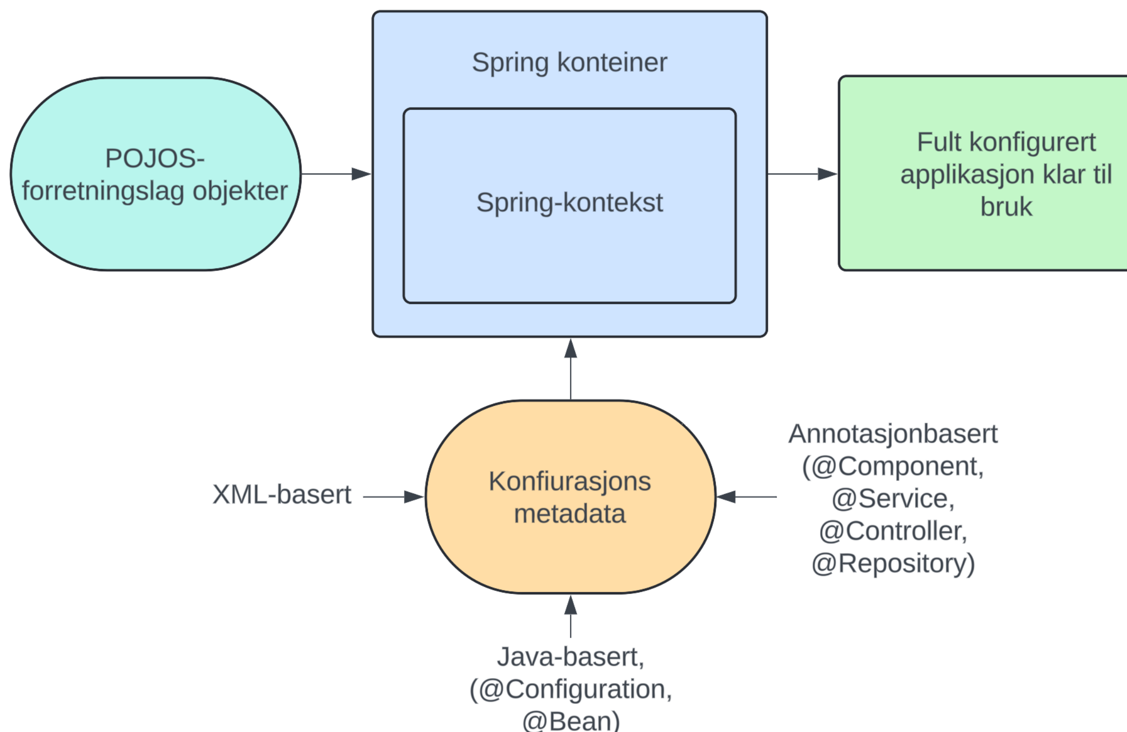
Kode eksempel 4.3: Eksempel på servicemetode som definerer en viktig del av forretningslogikken. (OrderService.java).

```
/**
     * Returns all registered Orders from getRegisteredOrders and sorts them
        into a Map of key-value pairs
     * grouped by both Store and wishedDeliveryDate.
     *
     * @return a Map where each key is a SimpleEntry of Store and
        wishedDeliveryDate, and each value is a list of Orders.
     * Example output:
     * [Store 1, 2024-04-15] -> [Order 1, Order 2]
     * [Store 1, 2024-04-16] -> [Order 3, Order 4]
     * [Store 2, 2024-04-16] -> [Order 5, Order 6, Order 7]
     */
    public Map<AbstractMap.SimpleEntry<Store, LocalDate>, List<Order>>
        groupByStoreAndDeliveryDate() {
        List<Order> registeredOrders = this.getRegisteredOrders();
        return registeredOrders.stream()
                .filter(order -> order.getCustomer() != null && order.
                    getCustomer().getStore() != null)
                .collect(Collectors.groupingBy(order ->
                        new AbstractMap.SimpleEntry<>(order.getCustomer().
                            getStore(), order.getWishedDeliveryDate())));
    }
```

4.3.5 Konfigurasjon

Konfigurasjonsklassene inneholder metoder som forvaltes gjennom Spring-konteksten gjennom hele kjøreperioden. Klassene definerer såkalte 'beans', eller bønner, gjen-

nom @Bean-annotasjonen, som blir håndtert av Spring-containeren når applikasjonen starter, og deretter injisert i Spring-konteksten for å levere nødvendig funksjonalitet til resten av applikasjonen. Andre annotasjonsbaserte konfigurasjonsdata som @Component, @Service, @Controller og @Repoistory blir også forvaltet og injesert på samme måte som bønner inn i Spring-konteksten. Figur 4.11 viser hvordan Spring-rammeverkets konfigurasjons- og oppstartsprosesser ser ut.



Figur 4.11: Spring-rammeverkets konfigurasjons- og oppstartsprosesser.

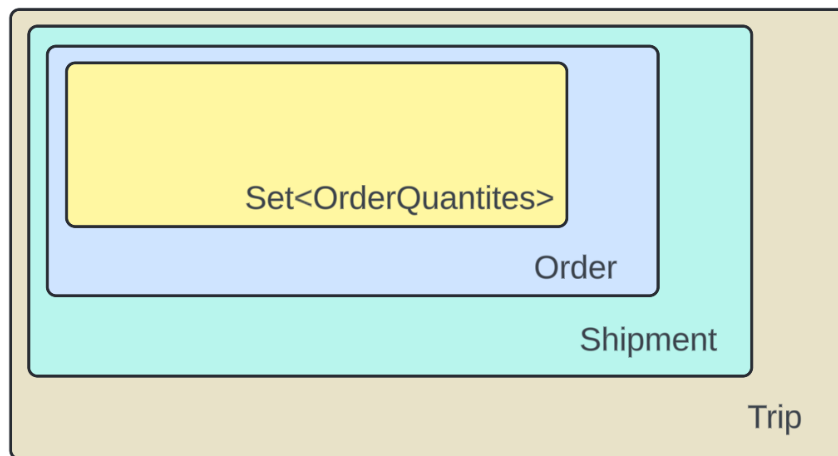
Bønner fungerer som er erstatning for tradisjonell XML-basert konfigurasjon for å tilby en mer intuitiv og programmatisk måte å konfigurere konteksten på. Konfigurasjonsklasser blir annotert med @Configuration for å indikere at klassen kan behandle bønner og bidra til konteksten som en konfigurasjonskilde. Bruken av bønner bidrar til lavere kopling og unngår unødvendig opprettelse av nye objekter når én enkelt instans er tilstrekkelig. Kode eksempel 4.4 viser hvordan bønner defineres på metodenivå.

Kode eksempel 4.4: Eksempel på en bønne som defineres på metode-nivå for å injesere konfigurasjonen for Swagger inn i Spring-kontektsen. (SwaggerConfig.java).

```
/**
     * Configures Swagger for the public APIs.
     *
     * @return a grouped OpenAPI definition for public APIs.
     */
    @Bean
    GroupedOpenApi publicApi() {
        return GroupedOpenApi.builder()
                .group("public-apis")
                .pathsToMatch("/**")
                .build();
    }
```

4.3.6 Database

Databasen er en MariaDB som er en 'fork', eller kopi, av den originale MySQL-databasen. Det er en relasjonsdatabase som organiserer data i tabeller og rader, hvor rader ofte blir referert til 'tupler' som inneholder dataen assosiert med en unik ID [61]. Den består av 9 tabeller gjennom ulike kardinaliteter, som vist i 4.2.4. Hovedlogikken i applikasjonen kommer fra mange-til-en kardinaliteten mellom 'trip', 'shipment', 'order' og 'orderquanitites'. Disse former strukturen til hvordan ordre blir plassert med ulike mengder varer, sortert og lagt til i en shipment, som blir lagt til i en trip, eller tur, som skal simulere en lastebil som leverer varene. Sammensetningen av databasen blir håndtert av Java Persistance API (JPA), som er et 'object-relation mapping (ORM) rammeverk som tilbyr mapping fra Java objekter til tabeller i en relasjonsdatabase [62]. JPA definerer et sett med annotasjoner som abstraherer bort de spesifikke detaljene ved databasehåndtering. Det har blitt tatt i bruk Hibernate som en implementasjon av JPA, som er et verktøy som støtter ORM-implementasjonen. Figur 4.12 viser til hvordan hovedstrukturen ser ut i et komposisjonsdiagram.



Figur 4.12: Mange-til-en (M:1) kardinalitet.

Annotasjoner som har blitt tatt i bruk for å definere databasestrukturen gjennom Hibternate er:

- **@Entity:** Brukes til mapping av Java klasser til en tabell i databasen. Annotasjonen blir ofte brukt med @Column for å definere kolonnene i tabellen.
- **@Id:** Brukes til å definere ID-en til tabellen. Ofte brukt med parameteret (strategy = GenerationType.IDENTITY) for å øke verdien av ID-feltet automatisk når en ny rad blir lagt til i tabellen.
- **@Table:** Brukes til å spesifisere detaljer om tabellen definert av @Entity. Annotasjonen kan ta i bruk flere parametre somn tabellnavn, schema og katalog.
- **@Column:** Brukes til å mappe felt definert i Java klassen til kolonner i tabellen. Annotasjonen har flere parametere for å definere detaljer og begrensninger om dataen som kolonnen kan inneholde. Eksempler er kolonnenavn, lengde, som begrenser antall karakterer, unik, som sikrer at alle dataene i kolonnen er unike, og nullable som er en boolean som sikrer om dataene kan være null eller ikke.

- **@ManyToOne:** Definerer et mange-til-en forhold mellom to entiteter. Annotasjonen sørger for at fremmednøkkelen på 'mange'-siden peker til primærnøkkelen på 'en'-siden.
- **@OneToMany:** Definerer et en-til-mange forhold mellom to entiteter. Annotasjonen sørger for at primærnøkkelen på 'en'siden refereres til av fremmednøkkelen på 'mange'-siden.
- **@OneToTone:** Definerer et en-til-en forhold mellom to entiteter. Annotasjonen sørger for at eieren av annotasjonen blir referert til av en tilsvarende fremmednøkkel i den relaterte entiteten.
- **@Enumerated:** Brukes til å definere at feltet i klassen skal opptre som en ENUM i kolonnen.
- **@Size, @Min, @Max, @NotBlank, @NotNull:** Brukes til å definere begrensinger ved dataene som blir lagt til i kolonnene.

4.3.7 Sikkerhet

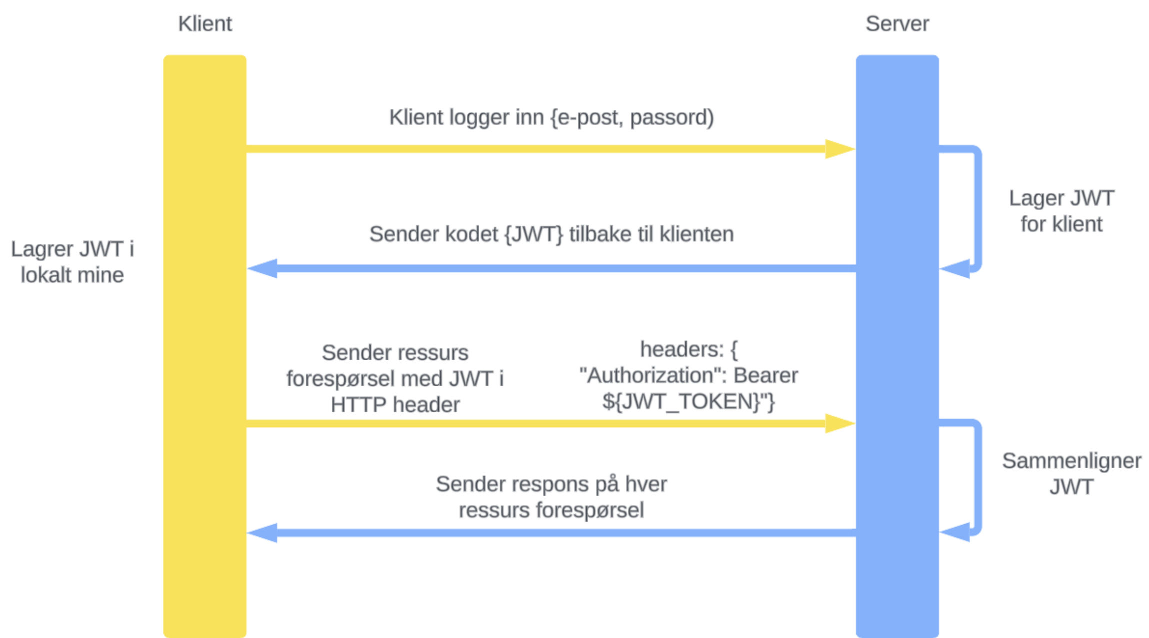
Ved å bruke Spring Security kan man begrense brukeres tilgang til kun de definerte endepunktene basert på deres rettigheter. Spring Security tilbyr en rekke implementasjoner som sørger for at brukeren må være autentisert og autorisert for å spørre etter, eller manipulere ressurser på endepunktene.

4.3.7.1 Brukerautentisering

I forbindelse med utviklingen av modulen som er ansvarlig for brukerautentisering ble det implementert JSON Web Tokens. Implementasjonen tar i bruk en 'auth0' mekanisme som generer JWT's for brukere, slik at de blir autorisert for tilgang til ressursene lagret i databasen gjennom API-et.

Når en bruker sender en HTTP forespørsel til /auth/login endepunktet om å logge inn, rutes forespørselen først gjennom en rekke filter i 'securityFilterChainen'. Dersom autentiseringen er gyldig, svarer serveren ved å returnere en kodet JWT i HTTP responsen. JWT-en som ligger som et objekt i responsen, lageres i brukerens lokale minne og det blir opprettet en 'session', eller økt mellom klienten og serveren.

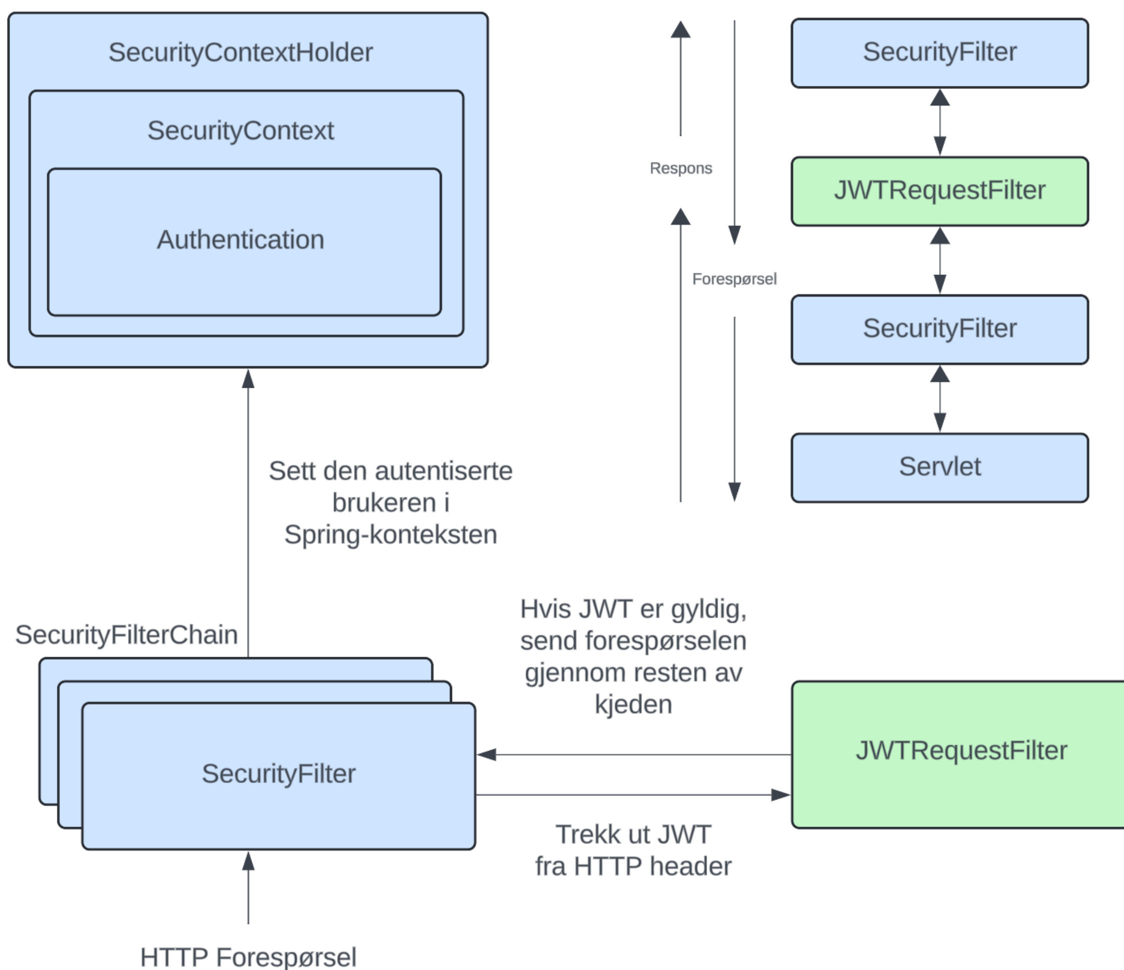
Figur 4.13 viser prosessen gjennom et sekvensdiagram fra at en bruker logger inn, til at serveren autentiserer og autoriserer ved hjelp av JSON Web Tokens.



Figur 4.13: JSON Web Token sekvensdiagram.

4.3.7.2 Sikkerhetskonfigurasjon

I Spring Security brukes 'SecurityFilterChain' for å tilby tilpassbare sikkerhetsfiltre som beskytter applikasjonens ressurser. SecurityFilterChainen kan konfigureres etter applikasjonens behov, og setter begrensinger for endepunkter som trenger autorisering. Hver gang en bruker logger inn, blir HTTP-forespørselen sendt gjennom en rekke filtre for å autorisere brukeren. Forespørselen blir behandlet og validert av securityFilterChain og JWTRequestFilter, et egendefinert autoriseringsfilter, for å sikre at brukerdetaljene er autentiske og har tilgang til de relevante ressursene. Figur 4.14 viser hvordan HTTP forespørselen sendes gjennom en rekke filter for å så sette brukeren i Spring-konteksten.



Figur 4.14: Flyten av en HTTP-forespørsel gjennom filterene.

Implementeringen av SecurityFilterChain og JWTRequestFilter stammer fra Spring Security for å konfigurerer sikkerhetsfiltre. JWTRequestFilter er en egendefinert klasse som arver fra 'OncePerRequestFilter', en klasse i Spring Security som er egnet for å lage egendefinerte filter. OncePerRequestFilter sørger for at filteret kun blir kjørt én gang per HTTP-forespørsel ved å implementere metoden doInternalFilter(). Dette gjør at JWTRequestFilterert kan integreres samløst i SecurityFilterChainen. Kode eksempel 4.5 og 4.6 viser implementasjonen av begge HTTP-filterene.

Kode eksempel 4.5: SecurityFilterChainen som jobber sammen med JWTRequest-Filter for å autentisere brukere for autoriserte endepunkt. (SecurityConfig.java).

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
    Exception {@Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity http)
            throws Exception {
        http
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtRequestFilter, AuthorizationFilter.class)
                    // Run authentication filter before http request filter
                .authorizeHttpRequests(authorize -> authorize.requestMatchers(
                    WHITELIST_DEVELOPMENT).permitAll() // Exclusion rules
                        .anyRequest().authenticated()); // Everything else
```

```
                        needs authorization
        return http.build();
    }
            http
                .csrf(csrf -> csrf.disable())
        .addFilterBefore(jwtRequestFilter, AuthorizationFilter.class)
        .authorizeHttpRequests(authorize ->
            authorize.requestMatchers(WHITELIST_DEVELOPMENT).permitAll()
            .anyRequest().authenticated());
    return http.build();
}
```

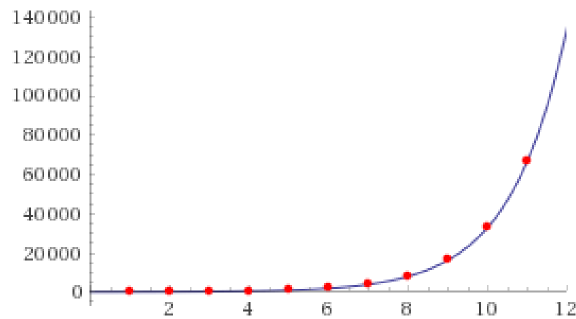
Kode eksempel 4.6: JWTRequestFilter som blir kjørt som et egendefinert filter i SecurityFilterChainen. (JWTRequestFilter.java).

```
@Override
    protected void doFilterInternal(HttpServletRequest request,
        HttpServletResponse response, FilterChain filterChain) throws
        ServletException, IOException {
        String tokenHeader = request.getHeader("Authorization"); // Extract JWT
             from header
        if(tokenHeader != null && tokenHeader.startsWith("Bearer ")) {
            String token = tokenHeader.substring(7);
            try {
                String email = jwtService.getEmail(token); // Compare JWT to
                    Customer with email
                Optional<Customer> opUser = customerRepository.findByEmail(
                    email);
                if(opUser.isPresent()){
                    Customer user = opUser.get();
                    UsernamePasswordAuthenticationToken authentication = new
                        UsernamePasswordAuthenticationToken(user, null, new
                        ArrayList()); // Create authentication object
                    authentication.setDetails(new
                        WebAuthenticationDetailsSource().buildDetails(request))
                        ;
                    SecurityContextHolder.getContext().setAuthentication(
                        authentication); // Set the authentication object to
                        the Spring security context
                }
            } catch (JWTDecodeException ex) {
            }
        }
        filterChain.doFilter(request, response); // Run defined filter
    }
```

4.3.7.3 Passordhashing med BCrypt

Som et sikkerhetstiltak er det blitt benyttet passordhashing med BCrypt for å sikre at passord ikke blir lagret som klartekst, eller kan reverseres til sitt opprinnelige form, som beskrevet i 2.3.2. BCrypt genererer en 16-byte (128 bit) randomisert streng, kalt 'salt', som er unik for hvert passord som skal lagres [63]. Dette betyr at selvom to brukere har samme passord, vil fortsatt saltet være tilfeldig og umulig å identi-

fisere at to passord er like i databasen. For å øke sikkerheten ytterligere, benytter også BCrypt en 'kostfaktor' for identifisere hvor mange iterasjoner som skal kjøres av hash-funksjonen. Kostfaktoren er basert på en logaritmisk eksponentiell skala, som betyr at tiden (ressursene) som kreves for å beregnet hashet, øker eksponentielt med kostfaktoren. Figuren 4.15 viser hvordan tiden vokser eksponentielt når kostfaktoren øker.

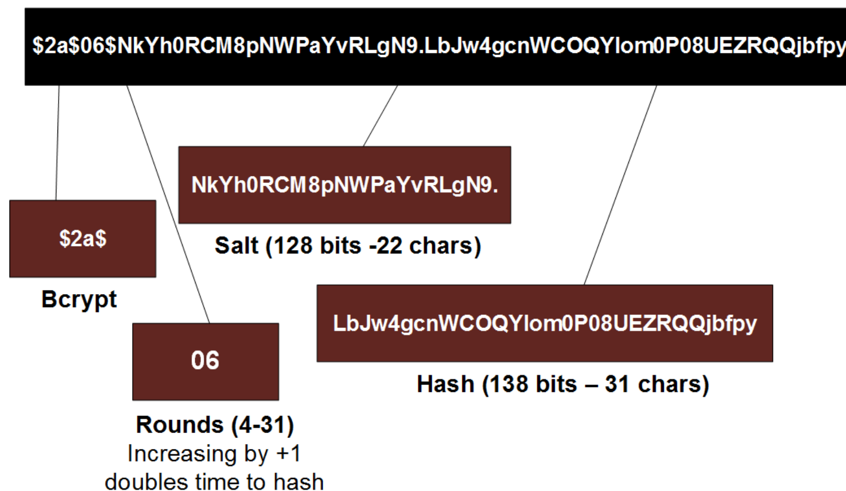


Figur 4.15: Eksempel på hvordan tiden (i millisekunder) vokser eksponentielt med antall iterasjoner definert i kostfaktoren.

Kilde: [63]

- For en kostfaktor på 10, vil antall interasjoner være 2^{10} = 1024.
- For en kostfaktor på 11, vil antall interasjoner være 2^{11} = 2048.
- For en kostfaktor på 12, vil antall interasjoner være 2^{12} = 4096.

Når hash-funksjonen er ferdig, har BCrypt generert et fullstendig hashet passord som inkluderer versjonen av BCrypt, antall iterasjoner i kostfaktoren, saltet og den hashede verdien. Figur 4.16 viser et eksempel på en fullstendig hashet streng som lagres i databasen.



Figur 4.16: Eksempel på komponentene den fullstendige hashede strengen består av.

Kilde: [64]

4.3.8 Infrastrucure as Code

Infrastrukturen som applikasjonen kjører på, er definert via Infrastructure as Code. IaC tillater å ha ressurser definert i kildekoden på en måte som både er versjonskontrollert og repeterbar. På denne måten reduseres manuelle prosesser i grensesnittet hos leverandøren ved å automatisere prosessen ved oppsettet og administrasjonen av infrastrukturen.

4.3.8.1 Terraform

Terraform har blitt tatt i bruk som et verktøy for å definere infrastrukturen på en deklarativ måte. Hver ressurs blir definert i en egen ressurs-blokk med nøkkelordet 'resource', med tilhørende konfigurasjonsparametre avhengig av hvilken ressurs som blir definert. Kodeeksempel 4.7 viser hvordan en virtuell maskin (VM) er definert for å kjøre applikasjonen.

Kode eksempel 4.7: Definering av virtuell maskin.

```
resource "google_compute_instance" "spring_boot_vm" {
  name         = "spring-boot-vm"
  machine_type = "e2-medium"
  zone         = var.GOOGLE_ZONE
  allow_stopping_for_update = true

  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2004-lts"
    }
  }
}
```

Den virtuelle maskinen tar inn en rekke konfigurasjonsparametre som navn, maskintype og hvilken sone verten skal være i. Nøkkelordet 'boot_disk' definerer hvilket operativsystemavbildning oppstartsdisken skal baseres på.

Videre er det også definert nettverksressurser som VPC (Virtual Private Cloud), subnettverk og brannmurregler for å tillate SSH-forbindelser over TCP på port 22 og vanlig internetttrafikk over TCP på port 8080. Kodeeksempel 4.8 viser hvordan brannmuren er konfigurert på VPC-nettverket for å tillate internetttrafikk.

Kode eksempel 4.8: Konfigurering av brannmur.

```
resource "google_compute_firewall" "spring_boot" {
  name          = "spring-boot-app-firewall"
  network       = google_compute_network.vpc_network.id
  target_tags   = ["spring-boot"]
  allow {
    protocol = "tcp"
    ports    = ["8080"]
  }
  source_ranges = ["0.0.0.0/0"]
}
```

Terraform integreres samløst med GitHub Actions for å etablere en CI/CD-pipeline.

4.3.8.2 CI/CD Pipeline med Github Actions

For kontinuerlig leveranse er en 'continuous integration/continuous delivery' (CI/CD) pipeline implementert ved hjelp av GitHub Actions. GitHub Actions benytter en workflow-fil av typen .yml til å definere de ulike jobbene som skal utføres i pipelinen. Pipelinen defineres av jobben 'build-and-deploy', som utfører en rekke trinn knyttet til den virtuelle maskinen. Disse trinnene inkluderer bygging og lagring av et Docker-image, Google Cloud-autentisering, opprettelse av en SSH-forbindelse, samt nedlasting og kjøring av det nyeste Docker-imaget. Disse stegene blir definert ved hjelp av Terraform's kommandolinjeverktøy som utfører oppgavene for å automatisere og vedlike infrastukturen. Nøkkelkommandoer som inisialiserer, planlegger og utfører ressursprovisjonering inkluderer:

1. Terraform init: Forbereder 'working directory', eller arbeidskatalogen, for andre kommandoer. Laster også ned nødvendige avhengigheter og konfigurerer tilstandsfilen lagret i en 'bucket' hos Google Cloud Storage.
2. Terraform plan: Lager en plan som sammenligner nåværende infrastruktur mot konfigurasjonen. Dersom det finnes endringer i konfigurasjonsfilen, visualiseres disse i pipelinen.
3. Terraform apply: Utfører planen laget i Terraform plan. Dersom planen ikke inneholder endringer, forblir infrastrukturen uendret.

Pipelinen kjører på hver 'commit' i main-branchen og tillater at serveren alltid kjører på den nyeste versjonen og at brukere alltid har tilgang til de nyeste dataene.

4.3.8.3 Enhetstesting i pipelinen

Enhetstestene blir definert som en separat job i workflow-filen, men som er avhengig av at build-and-deploy jobben er vellykket for å bli kjørt. Ved å inkludere enhetstester i pipelinen, kan vi raskt identifisere 'bugs' og validere at alle komponentene i systemet fungerer som forventet. Kode eksempel 4.9 viser hvordan jobben om enhetstester blir definert.

Kode eksempel 4.9: Eksempelet viser 'unit-tests' jobben som kjører enhetstestene i pipelinen. (terraform.yml).

```
unit-tests:
  runs-on: ubuntu-20.04
  needs: build-and-deploy
  steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up JDK 17
      uses: actions/setup-java@v2
      with:
        java-version: '17'
        distribution: 'adopt'

    - name: Cache Maven packages
```

```
        uses: actions/cache@v2
        with:
            path: ~/.m2
            key: ${{ format('{0}-m2-{1}', runner.os, hashFiles('**/pom.xml'))
                }}
            restore-keys: |
              ${{ runner.os }}-m2-

      - name: Install dependencies
        run: mvn -B dependency:resolve

      - name: Run tests
        run: mvn -B test
```

4.3.8.4 Hemmeligheter og variabler

CI/CD Pipelinen er avhengig av sensitiv informasjon for å fungere korrekt. Dette gjelder private SSH nøkler og autentiseringsdetaljer for ulike tjenester som er nødvendige for å få tilgang til eksterne ressurser og integrere med tredjepartstjenester. For å håndtere slik sensitiv informasjon, benyttes hemmeligheter og variabler i GitHub Secrets for sikker lagring og injisering i pipelinen når de trengs. På denne måten sikres det at sensitiv informasjon ikke er tilgjengelig i prosjektfilene. Kodeeksempel 4.10 viser hvordan autentiseringsdetaljene for databasen håndteres.

Kode eksempel 4.10: Konfigurasjon for databasen som henter verdier fra GitHub Secrets. (application.properties).

```
spring.datasource.url=${MYSQL_DATABASE_URL}
spring.datasource.username=${MYSQL_DATABASE_USERNAME}
spring.datasource.password=${MYSQL_DATABASE_PASSWORD}
```

4.3.9 Testing og testmiljø

Tester har blitt utført i et eget testmiljø, hvor det har blitt tatt i bruk en integrert H2 database for å emulere en produksjonsdatabase. H2-databasen benytter seg lokalt minne for å lagre data, noe som gir raske og effektive tester. På denne måten kjører testmiljøet helt isolert fra produksjonsmiljøet, som sikrer at testene ikke påvirker reelle data eller systemer. Kode eksempel 4.11 viser at ProductManagementSystemApplicationTests kjører med en 'test'-profil, som er en egen application-test.properties. Denne filen inneholder oppsettet for H2-databasen og Hibernate-konfigurasjoner.

Kode eksempel 4.11: Initialiserer spring-konteksten med 'application-test.properties' slik at testmiljøet bruker spesifikke konfigurasjoner for testing.

```
@SpringBootTest
@ActiveProfiles("test")
class ProductManagementSystemApplicationTests {
        @Test
        void contextLoads() {
        }
}
```

4.3.9.1 Rammeverk for testing

For testing har det blitt tatt i bruk en rekke rammeverk for å teste de ulike abstraksjonslagene. Siden abstraksjonslagene har ulike ansvarsområder, er det nødvendig å bruke rammeverk som passer til hvert spesifikt lag. Rammeverkene tilbyr også implementasjoner for mer effektive tester gjennom automatisering og mock-objekter.På denne måten sikres en hyppigere CI/CD-pipeline da alle enhetstester er integrert og kjøres automatisk ved hver commit. Det har blitt tatt i bruk rammeverk som:

- **Junit5:** Tilbyr et moderne fundament for enhetstesting gjennom robuste og dynamiske funksjoner [65].
- **Mockito:** Simplifiserer tester ved å 'mocke' avhengigheter for å skape bedre isolasjon i testmiljøet [66].
- **MockMVC:** Brukes til å sende false HTTP-forespørsler og verifisere HTTP-responser [67].

4.3.9.2 Intregrasjonstester

Det har blitt uført integrasjonstester for å teste at alle abstraksjonslag fungerer som de skal. Annotasjonen @SpringBootTest sørger for at hele spring-konteksten blir lastet inn, inkludert alle bønner og konfigurasjoner slik at testingen emulerer et virkelig produksjonsmiljø. Integrasjonstestene sørger for at:

1. Endepunktene returnerer dataen og responskoden som er forventet.
2. Interaksjonen mellom forskjellige lag i applikasjonen oppfører seg som forventet.
3. Databaseoperasjoner fungerer som forventet.
4. Konfigurasjoner og avhengigheter lastes inn korrekt.
5. Sikkerhetsmekanismer fungerer som forventet.

I integrasjonstester har rammeverket MockMVC blitt tatt i bruk, dette tilbyr ytterligere støtte for testing. MockMVC settes opp ved annotasjonen @AutoConfigureMockMvc og konfigurerer MockMVC objektet automatisk for bruk i tester. MockMVC utfører full Spring MVC-forespørselshåndtering, men via falske HTTP-forespørsels- og responsobjekter i stedet for en kjørende server [68]. På denne måten kan det sikres at endepunktene returner de riktige JSON-objektene og responskodene uten å være i produksjonsmiljøet. Kode eksempel 4.12 viser en integrasjonstest som tester '/auth/login' endepunktet i CustomerController ved å sende en HTTP Post-forespørselen med tilhørende brukerdetaljer.

Kode eksempel 4.12: Integrasjonstest som tester endepunktet som lar en bruker logge inn. (CustomerControllerIntegrationTest.java).

```
@Test
    public void testLoginSuccess() throws Exception {
        // Customer is created beforehand with a hashed password and saved to
           the CustomerRepository
```

```
        LoginBody validLogin = new LoginBody("test@example.com", "
            secretpassword11");
        String jsonRequest = objectMapper.writeValueAsString(validLogin);

        mockMvc.perform(post("/auth/login")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(jsonRequest))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.jwt").exists());
    }
```

4.3.9.3 Repositorytester

Datalaget har blitt testet med de egendefinerte SQL-spørringer for å sikre at disse fungerer som forventet. Repository-klassene er annotert med @DataJpaTest, som tilbyr en minimal Spring-kontekst for testing av persistens-laget. Kode eksempel 4.13 viser en repositorytest som tester en egendefinert SQL-spørring i OrderRepository klassen.

Kode eksempel 4.13: Testing av SQL-spørringen 'findByCustomer(Customer customer) som returnerer en liste med alle ordre assosiert til en bruker. (CustomerRepositoryTest.java).

```
@Test
    public void testFindByCustomer() {
        // Customer and Order is created beforehand and saved to their
            associated repository
        List<Order> orders = this.orderRepository.findByCustomer(customer);
        assertEquals(1, orders.size(), "There should be exactly one order for
            the customer");
    }
```

4.3.9.4 Servicetester

Forretningslogikken har blitt enhetstestet med rammeverket 'Mockito' for å sikre at alle metoder og regler i servicelaget fungerer som forventet, uten å være avhengige av eksterne systemer eller databaser. Mockito lager 'mock'-objekter, eller liksomobjekter, som simulerer oppførselen til virkelige objekter i testmiljøet. Annotasjonen @Mock brukes for å opprette mock-objekter, og @InjectMocks for å injisere mock-objektene inn i tjenesten som skal testes. Dette sørger for bedre isolasjon i tester, eliminerer avhengigheter til eksterne systemer og gjør det mulig å teste hyppigere siden hele konteksten ikke trenger å lastes inn [69]. Kodeeksempel 4.14 viser hvordan forretningslaget testes med liksomobjekter.

Kode eksempel 4.14: Testing av servicemetoden registerCustomer(Registrationbody registrationbody) i CustomerServiceTest (CustomerServiceTest.java).

```
@Test
    public void testRegisterCustomer_NewCustomer() {
        // Arrange
```

```
        RegistrationBody registrationBody = new RegistrationBody("test@example.
            com", "Test", "User", "password123", 1);
        when(customerRepository.findByEmailIgnoreCase(registrationBody.getEmail
            ())).thenReturn(Optional.empty();
        Store store = new Store();

        when(storeRepository.findById(registrationBody.getStoreId()).
            thenReturn(Optional.of(store));
        when(encryptionService.encryptPassword("password123")).thenReturn("
            encryptedPassword");

        // Act
        assertDoesNotThrow(() -> customerService.registerCustomer(
            registrationBody));

        // Assert
        verify(customerRepository).findByEmailIgnoreCase("test@example.com");
        verify(storeRepository).findById(1);
        verify(customerRepository).save(any(Customer.class));
    }
```

4.3.9.5 Endepunktstesting med Postman

Endepunktstesting for alle API-endepunkter ble utført med Postman. Postman er en applikasjon som tillater testing av API-er ved å visualisere og sende HTTP-forespørsler, samt motta og analysere HTTP-responser. Med Postman kunne alle endepunkter testes ved hjelp av ulike HTTP-verb, som beskrevet i avsnitt 2.1.3.1.

4.3.9.6 Belastningstest med Postman

Det har også blitt utført idempotente belastningstester på API-et for å undersøke hvordan responstiden påvirkes når antallet kunder som etterspør ressurser, øker. Testene viser tydelig at HTTP-responstiden øker betydelig når antallet samtidige brukere øker bare fra 15 til 50 brukere. Denne økningen i responstid kan knyttes opp til flere årsaker:

1. **Ressursbegrensinger på den virtuelle maskinen:** Når flere brukere sender forespørsler samtidig, øker belastningen på serverens prosessor og minne. Dette kan føre til køsystemer i planleggingen av prosesseringsoppgaver. Den virtuelle maskinen som serveren kjører på tilbyr kun 2 vCPU-er med delt kjerne og 4GB minne, men som kan skaleres etter behov.
2. **Tett integrerte komponenter:** Komponentene i den monolittiske arkitekturen er knyttet tett opp mot hverandre som betyr at inneffektivitet i én del av systemet kan påvirke hele applikasjonens ytelse.
3. **Flaskehalser i databasespørringer:** På grunn av den monolittiske arkitekturen som benytter en enkelt database, kan det oppstå flaskehalser når belastningen er høy.

Tabellen nedenfor oppsummerer resultatene fra tre ytelsestester utført med Postman. Testene ble gjennomført med forskjellige antall brukere i 2 minutter, som også var en 'ramp-up' periode med økende mengde brukere, som også betyr økende mengde HTTP-forespørsler.

Brukere	Tid	Tot. foresp.	Foresp./sek	Resp.tid (ms)
15	2 min ramp-up	2132	16.29	251
30	2 min ramp-up	3529	27.08	445
50	2 min ramp-up	3260	25.26	1115

Table 4.1: Oppsummering av Postman ytelsestester.

Resultatene viser at når antall brukere øker, øker også både antall totale forespørsler og forespørsler per sekund. Ved 15 brukere er API-et høyst effektivt med 16,29 forespørsler per sekund og en responstid på 251 ms. Ved 30 brukere øker antallet forespørsler per sekund til 27,08, men responstiden øker også til 445 ms. Når antallet brukere økes videre til 50, øker den gjennomsnittlige responstiden betraktelig (fra 445 ms til 1115 ms). Den lengre responstiden ved 50 brukere betyr at systemet bruker lengre tid på å svare på hver forespørsel, noe som resulterer i at færre forespørsler kan behandles i løpet av samme tidsperiode.

Resultatene fra ytelsestestene kan finnes som vedlegg under 'Postman ytelsestester' i vedlegg D.

4.3.10 REST API Dokumentasjon

REST API-et er dokumentert med 'Swagger', et verktøy for å dokumentere og visualisere API-er. Swagger tilbyr brukere et intuitivt grafisk grensesnitt i nettleseren med en oversikt over alle kontrollere med tilhørende endepunkter, inkludert detaljer om tilgjengelige HTTP-metoder, parametere og responsstrukturer.

4.3.11 Javadoc

Kildekoden er dokumentert med Javadoc for å kunne dokumentere klasser, metoder, grensesnitt og felt. Ved å bruke Javadoc kan man generere HTML-dokumentasjon som gir bedre oversikt og organisering. Ved å dokumentere kildekoden godt, blir den også lettere å vedlikeholde i fremtiden.

4.4 Front-end

Denne seksjonen vil ta for seg resultatet oppnådd i front-end, en moderne og funksjonell mobilapplikasjon. Målet er å gi leser et klart bilde av hvordan applikasjonen fungerer fra et brukerperspektiv, og argumentere for hvorfor den ser ut som den gjør. Det vil også bli beskrevet hvordan applikasjonen håndterer sikkerhet og hvordan den er koblet til back-end.

4.4.1 Brukergrensesnitt

Brukergrensesnittet til en applikasjon er avgjørende for hvordan brukeren opplever og samhandler med applikasjonen. I denne seksjonen vil det bli gitt en detaljert gjennomgang av applikasjonens brukergrensesnitt for å begrunne designvalg og funksjonalitet.

4.4.1.1 Design

Designet ble utviklet i tråd med oppdragsgivers ønsker og behov, og tilbakemeldinger og forslag fra oppdragsgiver ble integrert underveis. Fargepaletten ble tildelt av oppdragsgiver for å sikre at applikasjonen enkelt kunne tas i bruk og passe inn i deres eksisterende løsninger. Den iøynefallende gulfargen, som går igjen i hele applikasjonen, er oppdragsgivers hovedfarge, og alle andre komponenter følger også oppdragsgivers fargepalett. Det ble gjennomført tester for å sikre at fargekontrastene følger WCAG AA [70], for synlighet og tilgjengelighet. Ulike produktikoner har forskjellige farger for å representere ulike varetyper, tørrvarer er gule, frysevarer er blå, og kjølevarer er grønne, noe som hjelper brukerne med å raskt identifisere og forstå applikasjonens innhold. Illustrasjon av dette sees i figur 4.24. Viktige knapper og elementer har også blitt gitt den iøynefallende gule fargen.

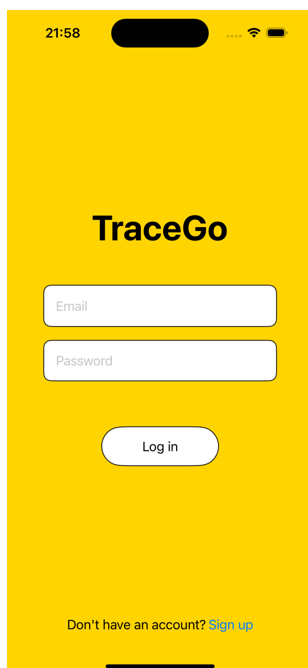
4.4.2 Funksjonalitet

I applikasjonen har det blitt implementert en rekke ulike funksjonaliteter for å møte oppdragsgivers ønsker. Alt fra å kunne logge inn på en egen bruker til å kunne skanne strekkoder for å lage en ny ordre. Nedenfor vil de ulike funksjonalitetene bli beskrevet.

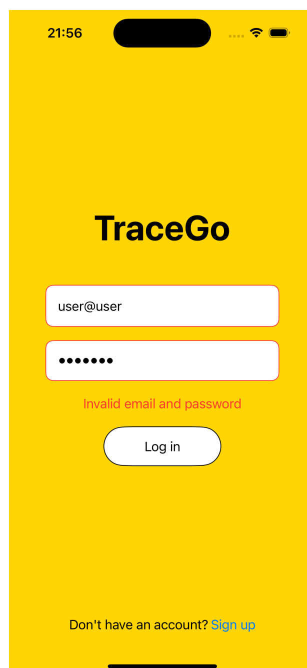
4.4.2.1 Login

Det første møtet med applikasjonen er innloggingssiden, som vises i figur 4.17. Her har brukeren mulighet til å logge inn med brukernavn og passord. Feltene hvor brukernavn og passord skal skrives inn kalles i Swift 'TextField' og 'SecureField'. Visuelt fremstår de to som identiske, og det er først når brukeren begynner å skrive at forskjellen blir klar. I SecureField vises ikke tegnene som blir skrevet inn på skjermen, og hvert tegn representeres heller av en prikk for å bevare sikkerheten [71].

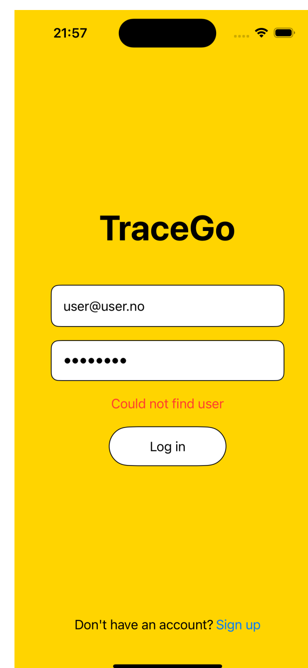
Når en bruker forsøker å logge inn, er det implementert feilhåndtering dersom det skulle oppstå problemer med enten inputen eller responsen fra server. Det benyttes regulære uttrykk for å sørge for at det skrives inn en gyldig e-post i tekstfeltet, og for at passordet følger visse krav. I figur 4.18 vises et eksempel på tilbakemelding til brukeren. I tillegg til tekst, visualiseres også hvor feilen er ved at det angitte feltet får en rød kant. I figur 4.19 kan man se et eksempel hvor brukeren har skrevet inn gyldig e-post og passord, men hvor brukeren ikke eksisterer.



Figur 4.17: Innlogging.



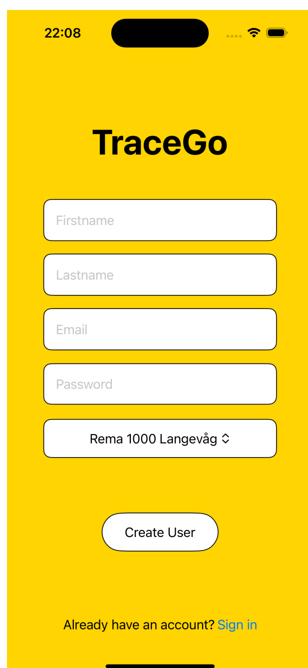
Figur 4.18: Innlogging med feil tilbakemelding.



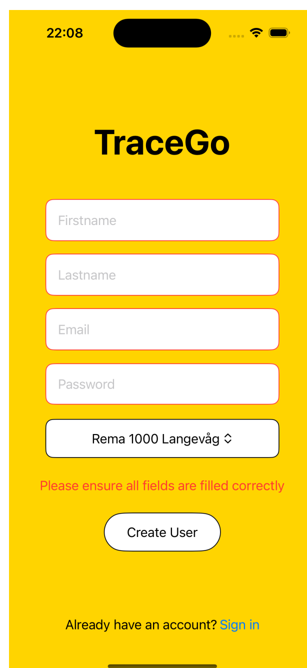
Figur 4.19: Innlogging når bruker ikke eksisterer.

4.4.2.2 Opprett bruker

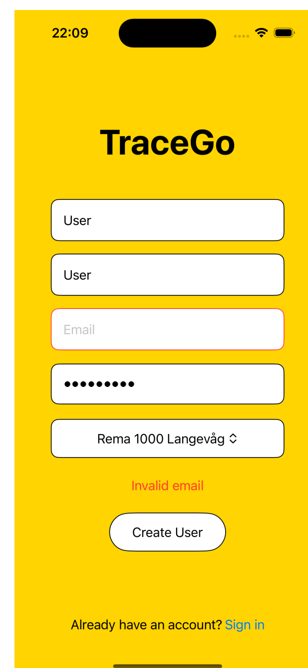
Dersom brukeren ikke allerede har en brukerkonto, kan man fra innloggingssiden navigere videre til en side som lar nye brukere opprette en konto. Denne siden kan sees i figur 4.20. Visuelt fremstår den veldig lik innlogginssiden, den eneste forskjellen at det er flere inputfelt for navn og valg av butikk. På samme måte som på innloggingssiden er det implementert feilhåndtering ved ugyldig input, hvor kanten på tekstfeltet som har feil blir rød i tillegg til feilmelding i form av tekst. Forskjellige eksempler på hvordan feilhåndteringen ser ut kan sees i figurene 4.21 og 4.22.



Figur 4.20: Registrer bruker.



Figur 4.21: Registrer bruker: Tomme felter.



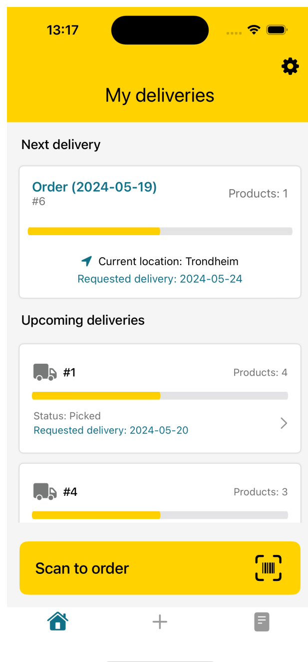
Figur 4.22: Registrer bruker: Manglende e-post.

4.4.2.3 Navigasjon

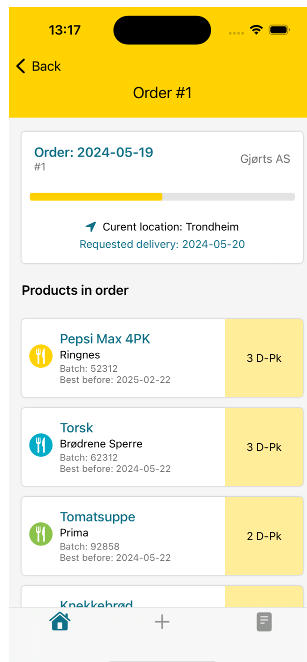
Etter suksessfull innlogging blir brukeren tatt til landingssiden, som gir informasjon om kommende leveranser og har en hurtigknapp for bestilling av varer ved hjelp av skanning av strekkode. Navigasjonslinjen gir tilgang til tre visninger: Hjem, Ny Ordre, Ordre Historikk. Innstillinger nås også fra landingssiden i verktøybaren øverst i høyre hjørne.

Som vist i figur 4.23, gir landingssiden en oversikt over neste planlagte leveranse med detaljer som bestillingsdato, antall forskjellige produkter og leveringsstatus. Brukeren kan se ordren sin nåværende plassering og ønsket leveringsdato. Kommende leveranser vises også med tilsvarende informasjon.

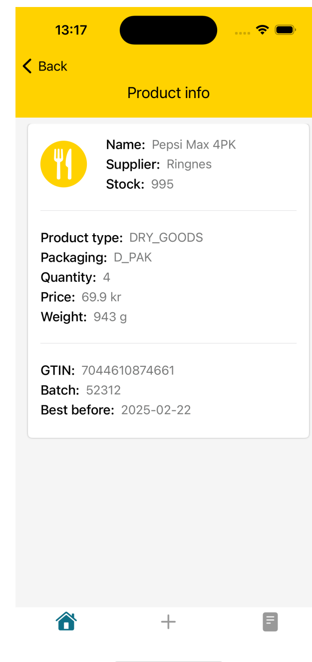
I figur 4.24 sees detaljer for en spesifikk ordre, med bestillingsnummer og produktdetaljer, blant annet annet hvor mange D-Pak man har bestilt av produktet. Denne siden nås ved å trykke på en ordre. Figur 4.25 viser produktinformasjon med detaljer som produktnavn, lagerbeholdning, kvantitet, pris, GTIN og best før-dato. Denne siden nås ved å trykke på et produkt.



Figur 4.23: Landingsside.



Figur 4.24: Ordre informasjon.

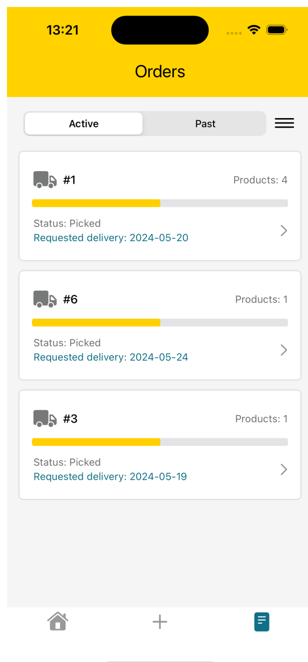


Figur 4.25: Produktinformasjon.

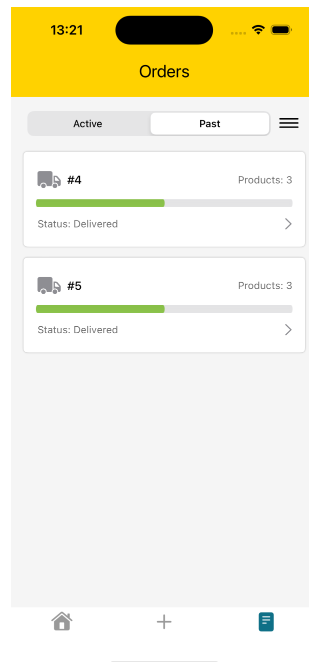
4.4.2.4 Ordre historikk

I ordrehistorikken kan brukeren se en liste av ordre. Ved hjelp av en 'picker', en kontroll fra SwiftUI-biblioteket, kan brukeren enkelt velge om de vil se aktive eller tidligere ordre [72]. I tillegg kan brukeren trykke på filterknappen ved siden av pickeren, som vil vise en 'halvside' med mulighet for å sortere ordre etter dato. Brukeren har valget mellom hurtigsortering eller å selv velge fra og til dato.

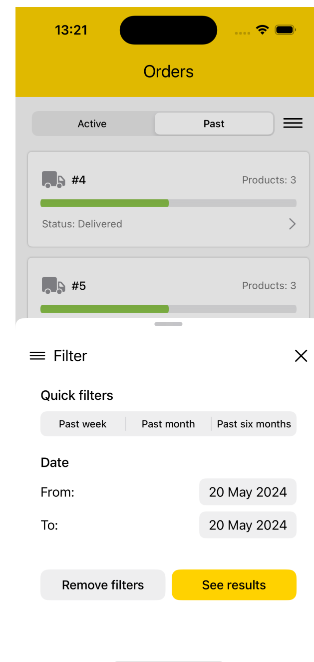
Som vist i figur 4.26, viser seksjonen for aktive ordre informasjon som ordrenummer, status og ønsket leveringsdato. I figur 4.27 vises listen over tidligere leverte ordre med tilsvarende informasjon. Filtreringsfunksjonen, som vist i figur 4.28, gir brukeren mulighet til å spesifisere et datointervall for å finne bestemte ordre.



Figur 4.26: Aktive ordre.



Figur 4.27: Leverte ordre.



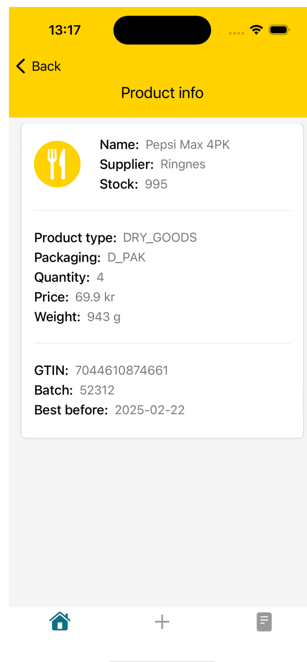
Figur 4.28: Filtrering av ordre.

4.4.2.5 Ordre

Ordresiden kan nås fra mange ulike elementer i applikasjonen, både gjennom ordrehistorikk og landingssiden med de ulike forventede leveringene. Ordrevisningen kan sees i figuren 4.24 og er beskrevet i avsnitt 4.4.2.3

4.4.2.6 Produkt

Fra ordresiden kan det navigeres videre inn til produktsiden (figur 4.29), hvor brukeren finner nødvendig informasjon om et spesifikt produkt. På toppen av siden vises informasjon som navn på produktet, navn på leverandør og varebeholdning hos leverandøren. Videre kommer informasjon om hvilken kategori produktet ligger i, informasjon om forpakning, antall produkter per pakke, pris og vekt. Til slutt kommer en oversikt over GTIN nummer, batch nummer og holdbarhetsdato.

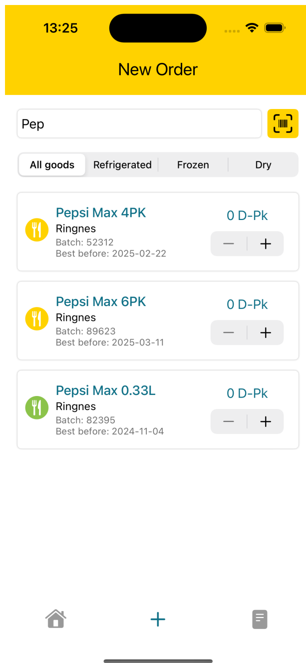


Figur 4.29: Informasjon om produkt.

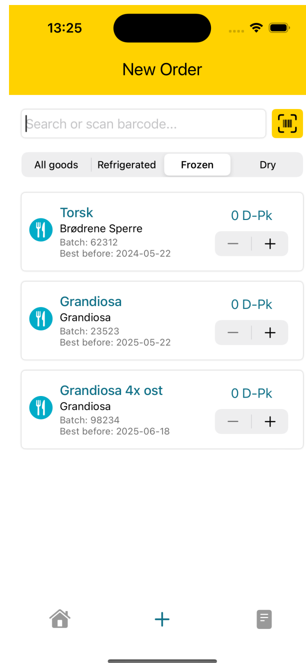
4.4.2.7 Ny ordre

På siden 'Ny ordre' kan brukeren legge til nye ordre de ønsker å bestille. Ordersiden viser en liste av ulike produkter med en stepper for å velge antallet man ønsker av et produkt. For å finne ønsket produkt har man tre muligheter: man kan søke etter produktet som vist i figur 4.30, filtrere på kategori som vist i figur 4.31, eller skanne strekkoden til produktet som vist i figur 4.32.

For å kunne implementere skanning av strekkode i applikasjonen har Apples egne rammeverk for kamera og mediehåndtering, AVFoundation, blitt brukt. Rammeverket gir tilgang til en rekke funksjoner for å fange og prosessere medieinnhold [50]. Når brukeren velger å skanne en strekkode, åpnes kameraet og viser hvor strekkoden skal plasseres. Applikasjonen sjekker om den har nødvendig tillatelse til å bruke kameraet og ber om det om nødvendig. Når en strekkode fanges opp, blir produktet identifisert og lagt til i handlekurven. Feilmeldinger håndteres hvis kameraet ikke er tilgjengelig eller strekkoden ikke kan leses. Et utdrag av koden finnes i kodeeksempel 4.15. Utdraget består av funksjonen som benyttes for å sjekke om applikasjonen har kameratillatelse.



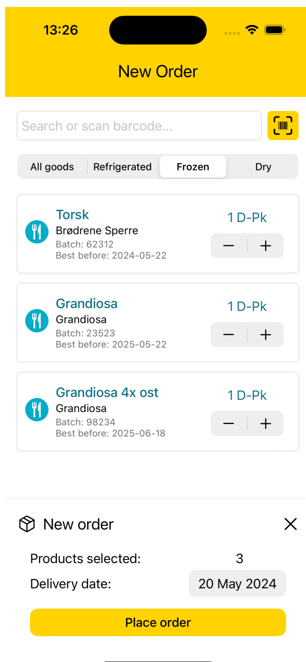
Figur 4.30: Søk etter produkt.



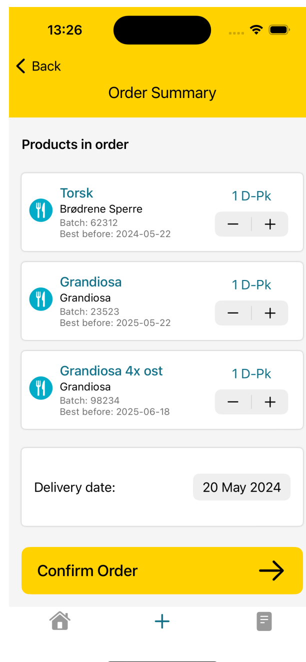
Figur 4.31: Filtrer etter produkttype.

Figur 4.32: Skann strekkode.

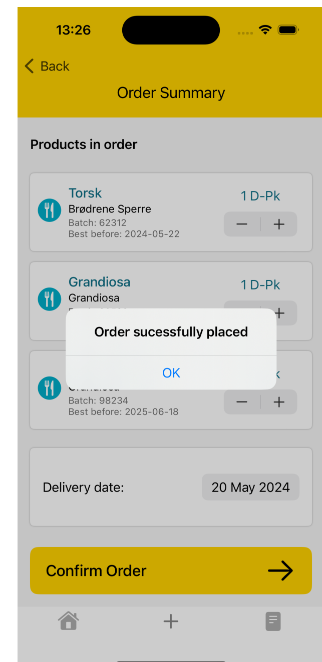
Når brukeren har lagt til alle ønskede produkter i bestillingen, trykker de på 'Place order' som vist i figur 4.33 og blir deretter tatt videre til en bekreftelsesside som vist i figur 4.34. Her kan brukeren gjøre siste endringer på produkter eller ønsket leverings dato, og deretter bekrefte ordren ved å trykke 'Confirm order'. Brukeren får da opp en melding med bekreftelse om at bestillingen er plassert (figur 4.35), og blir til slutt tatt til ordre siden med oversikten over den nye ordren.



Figur 4.33: Plasser ordre.



Figur 4.34: Rediger og bekreft ordre.



Figur 4.35: Ordre plassert.

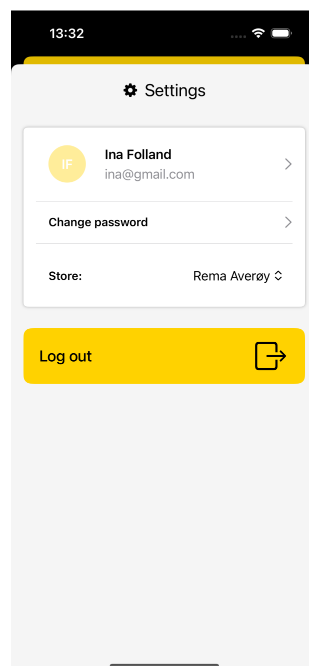
Kode eksempel 4.15: Sjekk av kamera tillatelse.

```
func checkCameraPermission (){
      Task{
          switch AVCaptureDevice.authorizationStatus(for: .video){
          case .authorized:
              cameraPermission = .approved
              setUpCamera()
          case .notDetermined:
              if await AVCaptureDevice.requestAccess(for: .video){
                  cameraPermission = .approved
                  setUpCamera()
              }else{
                  cameraPermission = .denied
                  showError("Please provide Access to Camera for scanning
                      barcode")
              }
          case .denied, .restricted:
              cameraPermission = .denied

          default: break
          }
      }
  }
```

4.4.2.8 Innstillinger

Innstillinger kan nås fra landingssiden ved å trykke på tannhjulet øverst i høyre hjørne. Her får brukeren mulighet til å endre informasjon som mailadresse, passord og hvilken butikk brukeren er tilknyttet. Det er også i innstillinger at brukeren kan logge ut av applikasjonen. Innstillinger kan sees i figur 4.36



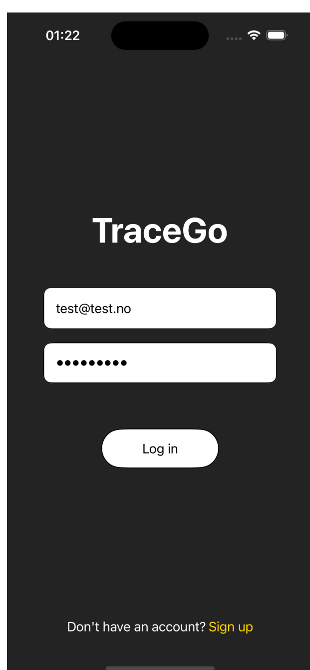
Figur 4.36: Innstillinger.

4.4.2.9 Mørk modus

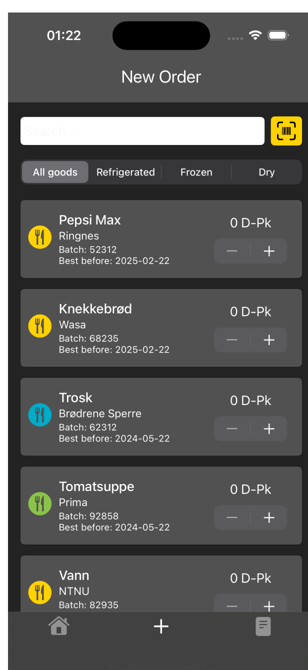
Applikasjonen har fått implementert støtte for både mørk modus og lys modus, slik at appens utseende automatisk tilpasses telefonens innstillinger. Dersom telefonen er satt til mørk modus, vises et mørkt fargetema, og ved lys modus brukes et lyst fargetema.

Mørk modus bruker en palett med dypere farger og høyere kontraster for å redusere øyebelastning og spare batteri. Bakgrunnene er mørke grå eller sorte, mens tekst og ikoner er lyse. Bilder av hvordan appen ser ut i mørk modus sees i figur 4.37, 4.38 og 4.39 Denne modusen er spesielt fordelaktig i svake lysforhold. Lys modus benytter en lysere palett med hvite bakgrunner og mørkere tekst og ikoner, som gir et klart utseende i dagslys og sterke lysforhold og fungerer best i lyse lysforhold.

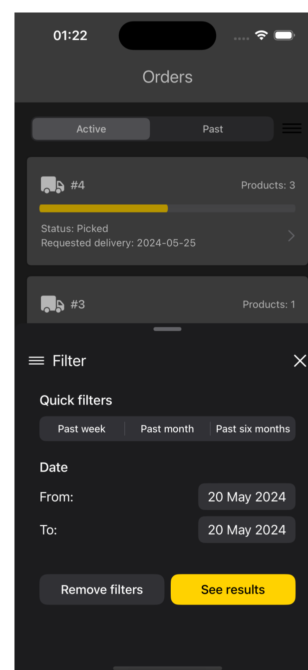
Funskjonaliteten for mørk og lys modus er implementert ved hjelp av SwiftUI Assets [73], som gjør det mulig å definere forskjellige fargetemaer som automatisk tilpasser seg systemets innstillinger.



Figur 4.37: Innloggingsside i mørk modus.



Figur 4.38: Ordrehistorikk i mørk modus.



Figur 4.39: Filtrering i mørk modus.

4.4.3 Sikkerhet

4.4.3.1 Keychain

Ved suksessfull innlogging i applikasjonen returneres det en autentiseringsnøkkel som benyttes for å hente informasjon fra back-end og verifisere brukertilgang, for eksempel ved registrering av en ny bestilling. Nøkkelen som brukes er en JWT-token (JSON Web Token) som beskrevet i avsnitt 2.3.1. Siden denne nøkkelen verifiserer brukeren og gir tilgang til sensitive data på serveren, er det avgjørende at den lagres sikkert for å forhindre uautorisert tilgang. For å oppnå dette har det blitt implementert en

KeychainManager-klasse for sikker lagring av autentiseringsnøkkelen.

KeychainManager-klassen bruker Keychain, en sikker lagringsmekanisme av apple, for å beskytte sensitiv informasjon som passord og tokens [74]. Klassen inneholder metoder for å lagre, hente og slette tokens basert på e-postadresser. Ved lagring slettes eventuelle eksisterende tokens med samme e-postadresse for å unngå duplikater. Henting og sletting utføres sikkert ved hjelp av Keychains innebygde funksjoner. Keychain er en del av Security, et rammeverk av apple, beskrevet i avsnitt 3.4.2.6. Kodekesmpel 4.16 viser et utdrag av KeychainManager klasse med metoden saveToken.

Kode eksempel 4.16: Lagring av token ved bruk av Keychain.

```
func saveToken(_ token: String, for email: String) throws {
        let tokenData = Data(token.utf8)
        let query: [String: Any] = [
            kSecClass as String: kSecClassGenericPassword,
            kSecAttrAccount as String: email,
            kSecValueData as String: tokenData
        ]

        let deleteStatus = SecItemDelete(query as CFDictionary)
        if deleteStatus != errSecSuccess && deleteStatus != errSecItemNotFound
            {
            throw KeychainError.failedToDelete
        }

        let addStatus = SecItemAdd(query as CFDictionary, nil)
        if addStatus != errSecSuccess {
            throw KeychainError.failedToSave(status: addStatus)
        }
    }
```

4.4.4 REST API kommunikasjon

For at applikasjonen skal kunne kommunisere med back-end er det implemntert et REST-API på serveren som mobilapplikasjonen kommuniserer med ved bruk av HTTP. Ved brukerinnlogging sendes en POST-forespørsel fra front-end til serveren med brukerens e-post og passord i JSON-format. Når serveren validerer brukeren, returneres en JWT-token (JSON Web Token) som lagres i Keychain på enheten. Etter vellykket innlogging henter applikasjonen brukerens detaljer ved å sende en GET-forespørsel til serveren med tokenen inkludert i Authorization-headeren. Serveren returnerer brukerdata i JSON-format, som dekodes og lagres i applikasjonens minne. Tokenen lagres i Keychain for tilgjengelighet etter innlogging og hentes derfra ved behov. Ved opprettelse av ny bruker sendes en POST-forespørsel med brukerens detaljer til serveren, og den nye brukeren logges automatisk inn etter registrering.

I tillegg til autentisering har det blitt implementert funksjoner for å hente ordrer, hente produkter og opprette nye ordrer. For å hente ordrer fra serveren, sendes en GET-forespørsel til et endepunkt med autentiseringstokenen. Tilsvarende sendes en GET-forespørsel for å hente produktinformasjon fra serveren. Når en ny ordre skal opprettes, sendes en POST-forespørsel med ordredata i JSON-format til serveren, hvor dataene blir behandlet og lagret.

Kodeksempel 4.17 viser hvordan applikasjonen henter ordre fra back-end. Her har det blitt benyttet 'TaskGroup' for å forbedre ytelsen ved parallell behandling, slik at nåværende lokasjon for hver ordre hentes samtidig som ordren, i stedet for sekvensielt. Dette halverte den totale tidsbruken på å laste ned ordre. TaskGroup har blitt brukt i kombinasjon med en aktør (OrderStorage), som sørger for trådsikker tilgang til data [75].

Kode eksempel 4.17: Henting av ordre fra back-end.

```
func fetchOrders(token: String) async throws {
    DispatchQueue.main.async {
        self.orders.removeAll()
    }

    guard let url = URL(string: "http://35.246.81.166:8080/api/orders")
        else {
        print("Invalid URL")
        throw NSError(domain: "Invalid URL", code: 0, userInfo: nil)
    }

    let headers = [
        "content-type": "application/json",
        "authorization": "Bearer \(token)"
    ]

    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.allHTTPHeaderFields = headers

    do {
        let (data, response) = try await URLSession.shared.data(for:
            request)
        guard let httpResponse = response as? HTTPURLResponse,
              (200...299).contains(httpResponse.statusCode) else {
            print("Error with the response, unexpected status code: \(
                String(describing: response))")
            throw NSError(domain: "Response Error", code: 0, userInfo: nil)
        }

        let orders = try JSONDecoder().decode([Order].self, from: data)

        let storage = OrderStorage()

        await withTaskGroup(of: Void.self) { group in
            for order in orders {
                group.addTask {
                    let currentLocation: String?
                    do {
                        currentLocation = try await self.getCurrentLocation
                            (orderId: order.orderId, token: token)
                    } catch {
                        print("Error fetching current location for order \(
                            order.orderId): \(error)")
                        currentLocation = nil
                    }
```

```
                let updatedOrder = order
                updatedOrder.currentLocation = currentLocation
                await storage.append(updatedOrder)
            }
        }
    }

    let updatedOrders = await storage.orders

    DispatchQueue.main.async {
        self.orders.append(contentsOf: updatedOrders)
    }

} catch {
    print("Error: \(error)")
    throw error
}
}
```

Del 5

Diskusjon

5.1 Prosessen

I denne seksjonen vil prosessen gjennom prosjektperioden diskuteres. Dette innebærer kommunikasjon, planlegging og tidsbruk, og hvordan scrum-metodikken har bidratt til en god arbeidsflyt gjennom iterativt arbeid.

5.1.1 Kommunikajson

Gjennom hele prosessen har intern kommunikasjon vært en prioritet for gruppen. Det har vært regelmessig kontakt både fysisk og digitalt via Discord. Dette sikret at alle gruppemedlemmer til enhver tid var oppdaterte på hverandres arbeidsoppgaver. Det ble benyttet en kombinasjon av fysisk arbeid på campus og kontoret, i tillegg ble hjemmekontor benyttet, hvor kommunikasjonen foregikk gjennom Discord. Denne fleksible tilnærmingen har vist seg å være svært effektiv. Når det var behov for diskusjoner eller tett muntlig samarbeid valgte gruppen å arbeide sammen på campus. Ved behov for assistanse fra oppdragsgiver benyttet gruppen kontoret.

5.1.1.1 Oppdragsgiver

Kommunikasjonen med oppdragsgiver har vært veldig god gjennom hele prosjektet. Muligheten til å dele kontorlokaler med gruppens kontaktpersoner hos Solwr, har gjort det mulig å raskt adressere og løse utfordringer som har oppstått. Dette har også gitt verdifull tilgang til veiledning og faglige råd.

5.1.2 Planlegging og tid

I forbindelse med forprosjektplanen ble det utarbeidet et Gantt-diagram for å planlegge forventet fremdrift. Gjennom prosjektperioden ble det imidlertid tydelig at denne tidsplanen var urealistisk for gruppen å følge. Det største tidsavviket fra diagrammet, som betydelig påvirket prosjektets progresjon, skyldtes en undervurdering av mengden arbeid i andre fag. Dette, sammen med en undervurdering av den nødvendige tidsbruken for opplæring i et nytt programmeringsspråk, resulterte i at kursing i Swift og SwiftUI ikke ble fullført før i midten av april.

5.1.2.1 Scrum

I prosjektets innledende fase, fra uke 3 til uke 12, ble det benyttet én ukes sprinter. Denne periodiseringen fungerte greit i startfasen, men det ble etter hvert tydelig at teamet ikke alltid klarte å avslutte åpnede saker fra backlogen før sprintens avslutning. Som en respons på denne utfordringen, ble det i etterkant av den syvende sprinten (uke 12) besluttet å forlenge sprintlengden til to uker. Dette tiltaket førte til en mer ryddig prosess, og det ble enklere å utarbeide grundige sprint reviews og retrospektiver, ettersom det var to ukers arbeid å evaluere, sammenlignet med én. I retrospekt ser teamet at det ville vært fordelaktig å implementere to ukers sprinter fra prosjektets begynnelse, da dette formatet tillot en mer strukturert tilnærming og forenklet planleggings og evalueringsprosessen.

Møtereferater fra sprintplanlegging og evaluering finnes som vedlegg under 'Prosjekthåndbok' i vedlegg B.

5.2 Resultatet

5.2.1 Mobilapplikasjonen

5.2.1.1 Valg av programmeringsspråk

Som nevnt i avsnitt 3.4.1.2 valgte gruppen å bruke Swift for å utvikle front-end delen av prosjektet. Veilederne ønsket en iOS-applikasjon fremfor en multiplattform-løsning med Flutter, og gruppen så på dette som en mulighet til å lære noe nytt i tillegg til å bruke eksisterende kunnskaper opparbeidet gjennom de tre årene med studier. Opprinnelig var det planlagt å bruke Flutter, et programmeringsspråk som deler av gruppen hadde erfaring med fra tidligere kurs. Swift og Flutter har begge sine fordeler og ulemper. Flutter er en multiplattformløsning som tilbyr et rammeverk for å bygge applikasjoner for flere plattformer fra en enkelt kodebase [76]. Dette ville gjort det mulig å ha en applikasjon som kunne kjøre på både iOS og Android, men oppdragsgiver argumenterte for at Swift generelt har bredere støtte i utviklermiljøet og at Flutter kanskje aldri vil bli så stort som ønskelig. Derfor ble det bestemt at en ren iOS-applikasjon var det beste valget.

Erfaringene gruppen har gjort seg med Swift er generelt svært positive. Det er et moderne og enkelt språk, med mye god dokumentasjon og mange verktøy. Dette ble veldig tydelig etter hvert som vi arbeidet mer med det. Apple har mange rammeverk som enkelt tillater implementasjon av kompleks funksjonalitet, som for eksempel strekkodeskanning. Rammeverkene som har blitt tatt i bruk i dette prosjektet inkluderer AVFoundation og Security. Et rammeverk som gjerne skulle ha blitt tatt i bruk er Core Data. I avsnitt 3.4.2.7 diskuteres dette rammeverket og hvorfor det gjerne skulle ha blitt benyttet.

5.2.1.2 Design

Designprosessen startet i Figma, hvor det ble utviklet wireframes i samarbeid med oppdragsgiver. Figma ble valgt fordi det tillater rask iterasjon og visuell kommunikasjon av designideer, noe som sikrer enighet om designet før koding begynner og sparer tid og ressurser.

Fargekontrastene i applikasjonen ble sjekket for å sikre at de følger WCAG AA, noe som gjør applikasjonen mer tilgjengelig for alle brukere. Kontrastkravet mellom tekst og bakgrunn er på minimum 4.5:1. I applikasjonen er den laveste kontrasten på 4.95:1 mellom hvit bakgrunn og grå tekst, og den høyeste på 14.63:1 mellom gul og svart. Farger ble også brukt strategisk for funksjonalitet. Produktikoner har forskjellige farger for å representere varetyper, og viktige elementer er gule for blikkfang.

Videre har designprinsippene som nevnt i avsnitt 2.6 blitt brukt under utviklingen av designet. Prinsippet om konsistens har blitt anvendt ved å gjenbruke elementer i designet og sikre et sammenhengende design i hele brukergrensesnittet med like og lignende utformede komponenter. Dette skaper en forutsigbar og trygg brukeropplevelse, hvor brukerne raskt kan bli kjent med og navigere applikasjonen. Prinsippet om tilbakemelding er ivaretatt ved at brukeren alltid får en visuell respons når en komponent trykkes på eller en handling utføres, for eksempel gjennom feilmeldinger eller bekreftelser. Dette bidrar til å informere brukerne om statusen på deres handlinger og forbedrer interaksjonen med applikasjonen. Prinsippet om begrensninger har også vært sentralt, med fokus på at brukeren kun skal ha tilgang til det nødvendige, og ikke mer. Dette reduserer kompleksiteten og gjør det lettere for brukerne å fokusere på de viktigste funksjonene.

Disse prinsippene og strategiene bidro til å skape en brukervennlig og tilgjengelig applikasjon som oppfyller oppdragsgivers krav og gir en god brukeropplevelse.

5.2.1.3 Funksjonalitet

Applikasjonen implementerer viktige funksjoner som innlogging, registrering, navigasjon, ordrebehandling og støtte for mørk modus. Innloggings- og registreringssidene er brukervennlige og sikre, med visuell tilbakemelding ved feil for å sikre korrekt input. Navigasjonen er intuitiv, med en tydelig navigasjonslinje som gir rask tilgang til hovedfunksjonene.

Ordrehistorikken er godt strukturert med filtreringsmuligheter, noe som gir fleksibilitet for brukerne. Funksjonaliteten for å opprette nye ordrer er også brukervennlig, med alternativer for søk, filtrering og strekkodeskanning via AVFoundation. Videre gir innstillingssiden mulighet for å oppdatere personlig informasjon og logge ut. Støtten for både mørk og lys modus forbedrer brukeropplevelsen ved å tilpasse utseendet etter brukerens preferanser og lysforhold. Bruken av SwiftUI Assets sikrer en moderne og dynamisk applikasjon.

For å forbedre applikasjonen ytterligere, kunne push-varsler implementeres for å varsle brukerne om viktige hendelser i sanntid. I tillegg ville en offline-modus la brukerne arbeide uten internettforbindelse med automatisk synkronisering når tilkoblingen gjenopprettes. Implementasjonen av en kartfunksjon for sanntidsoppdatering av leveranser ville gitt en visuell oversikt over leveransens plassering, øke effektiv-

iteten og redusere usikkerheten.

Applikasjonen er allerede robust, sikker og brukervennlig, med moderne rammeverk og teknologier. Likevel kan videre forbedringer som push-varsler, offline-modus og kart for leveranser øke verdien og brukervennligheten ytterligere.

5.2.1.4 Sikkerhet

Applikasjonen bruker JSON Web Tokens (JWT) for autentisering, og lagrer disse sikkert ved bruk av Keychain. Dette sørger for at brukernes identitet og tilgangsrettigheter er beskyttet mot uautorisert tilgang. Ellers lagres foreløpig all annen data som kun er nødvendig under kjøring av applikasjonen i minnet uten kryptering. Selv om operativsystemet beskytter minnet ved å isolere applikasjoner og implementere adgangskontroll, ville det likevel være ønskelig å kryptere annen brukersensitiv data, som e-postadresser og navn, for å gi et ekstra lag av sikkerhet.

Hvis Core Data senere blir implementert for lokal datalagring, vil kryptering av data være ønskelig. Kryptering av data i Core Data kan utføres ved å utnytte Keychain i Apples Security-rammeverk, noe som vil gi ekstra beskyttelse og sikre at dataene forblir konfidensielle og beskyttet mot uautorisert tilgang.

5.2.1.5 Integrasjon med Serverapplikasjonen

Som beskrevet i avsnitt 4.4.4 benyttes HTTP for kommunikasjon mellom front-end og back-end. En rekke forespørsler sendes fra front-end til back-end, der data knyttet til brukere beskyttes ved bruk av token-autentisering. Dette sikrer at sensitive brukerdata håndteres sikkert. Forespørslene kjører på egne tråder adskilt fra hovedtråden, noe som forhindrer at hovedtråden blokkeres og bidrar til en mer responsiv applikasjon.

Kjøring av flere tråder og gjentatte kall for å hente inn data kan imidlertid føre til noe innlastningstid for applikasjonen. Dette kunne ha blitt forbedret ved bruk av Core Data, som muliggjør lokal lagring og rask tilgang til data. Likevel ble det gjort forbedringer hvor mulig, for eksempel ved bruk av TaskGroup for parallell behandling, noe som halverte nedlastingstiden for ordrer ved å hente flere datasett samtidig.

Feilhåndtering er ivaretatt ved hjelp av ulike HTTP-statuskoder i responsene, som gir klar tilbakemelding til brukerne ved problemer. Videre forbedringer kan inkludere caching-mekanismer, som vil redusere antall forespørsler til serveren og forbedre ytelsen, samt bruk av websockets for sanntidsoppdateringer. Implementasjonen av websockets diskuteres videre i avsnitt 5.2.2.3.

Samlet sett er integrasjonen med backend effektiv og robust, med moderne autentiseringsmetoder, parallell behandling og gode feilhåndteringsmekanismer som sikrer en rask, pålitelig og sikker applikasjon..

5.2.2 Serverapplikasjon

5.2.2.1 Valg av systemarkitektur

Den opprinnelige planen var å utvikle en mikrotjenestearkitektur som kunne integreres med Solwr's eksisterende systemer. Siden Solwr allerede har mikrotjenester som håndterer alle ordre- og transporttjenester, kunne vi ha benyttet deres API-er for å integrere vårt system. Dette ville ha resultert i et mer løst koblet baksystem, ettersom det ikke ville vært nødvendig å utvikle funksjonalitet for ordre- og transporttjenester. Applikasjonen hadde også vært mer lettvektig og lettere å vedlikeholde. Etter diskusjon med Solwr-teamet ble det imidlertid klart at dette ikke var mulig, da deres systemer var for kompliserte og ville krevd omfattende arbeid for å implementere.

Derfor ble det i stedet utviklet et monolittisk system som inneholder moduler for både ordre- og transporthåndtering, men med mock-data. Ideelt sett ville arkitekturen i systemet vært betydelig annerledes, da det egentlig krever flere applikasjoner som samhandler om transport for å kunne implementere virkelige data fra lastebilens kjøretur. Dette hadde ført til sanntidssporing av pakker og en mer nøyaktig og effektiv håndtering av logistikkoperasjoner.

Den tiltenkte systemarkitekturen kan sees på figur 2.9.

5.2.2.2 Spring Cloud Gateway

Som en del av den opprinnelige planen var det tiltenkt å utnytte Spring Cloud Gateway, som fungerer som en selvstendig applikasjon. Denne gatewayen tilbyr funksjonaliteter som caching, HTTP-forespørselsbegrensninger, multitråding og ruting til riktig mikrotjeneste gjennom 'predikater'. Ved å benytte Spring Cloud Gateway ville mobilapplikasjonen ha opplevd raskere responstider, da brukerne kunne kommunisere direkte med gatewayen i stedet for baksystemet, på grunn av den innebygde cachen. Dette ville også ha resultert i bedre enkapsulering av systemet, ettersom kommunikasjonen ville ha vært begrenset til gatewayen. I tillegg støtter Spring Cloud Gateway overvåking, noe som kunne ha bidratt til forbedret drift og vedlikehold av systemet.

5.2.2.3 Implementasjon av Websockets

For å forbedre brukeropplevelsen kunne Websockets blitt implementert for sanntids- og hendelsesdrevet kommunikasjon mellom klienten og serveren. Vanlig kommunikasjon over HTTP gjennom TCP tilbyr kun en syklusdrevet modell til forbindelsen er terminert, mens Websockets gir en kontinuerlig toveisforbindelse gjennom en dedikert kanal, som muliggjør raskere og mer effektiv datautveksling [77]. På denne måten ville brukere kunne fått sanntidsoppdateringer i mobilapplikasjonen som gjelder pakkesporing og oppdateringer på produkter.

5.2.2.4 Det potensielle fullstendige systemet

Det potensielle fullstendige systemet ville benytte en mikrotjeneste arkitektur, bestående av flere selvstendige tjenester som håndterer transport, ordre, produkter og brukere. Denne tilnærmingen ville resultere i en samling løst koblede tjenester som kjører i egne containere i isolerte miljøer. Ved å implementere en slik arkitektur, ville hver mikrotjeneste være mer lettvektig siden de ikke trenger å inneholde informasjon som andre mikrotjenester håndterer. Mikrotjenester er også enklere å vedlikeholde fordi de kun er ansvarlige for en spesifikk del av systemet, i motsetning til å håndtere hele systemet.

Dette gir høy skalerbarhet, både horisontalt og vertikalt. Horisontal skalerbarhet betyr at antallet instanser av en mikrotjeneste kan økes for å håndtere økt trafikk. Vertikal skalerbarhet innebærer at ressursene som CPU, minne og lagring kan økes for en enkelt instans for å forbedre ytelsen. Ved å ta i bruk disse skaleringsmetodene, kan systemet effektivt tilpasses ulike belastningsnivåer og ressursbehov.

For å administrere containerinstansene av mikrotjenestene, ville det vært ideelt å bruke Kubernetes for kontainerorkestrering. Kubernetes automatiserer oppgaver knyttet til kontaineradministrasjon og tilbyr innebygde kommandoer for effektiv distribusjon og utrulling av endringer i mikrotjenestene. Kubernetes tilbyr også automatisk skalering, både horisontalt og vertikalt, gjennom overvåkning, noe som gjør administrasjon og vedlikehold enklere [78].

På den andre siden hadde en slik mikrotjeneste arkitektur vært krevende å vedlikeholde. På grunn av dens enorme kompleksitet ville det vært nødvendig å ha et team med riktig ekspertise innenfor ledelse, DevOps og utvikling.

5.3 Videre arbeid

Under prosjektperioden var det mange ulike funksjonaliteter gruppen ønsket å implementere i applikasjonen. Mange av de er implementert og beskrevet i del 4, men på grunn av begrenset tid er det også en del som ikke er implementert, men som gruppen mener kunne bidratt til å forbedre applikasjonen og brukeropplevelsen.

5.3.1 Push notifkasjoner

Oppdragsgiver uttrykte et ønske om at endringer i leveringsstatus skulle utløse push-notifikasjoner, slik at brukerne alltid kan se den nyeste statusen uten å måtte åpne applikasjonen og huske å følge med der. Firebase tilbyr løsninger for push-notifikasjoner som kunne ha blitt brukt for å implementere denne funksjonaliteten [79]. Siden Firebase ikke ble brukt i det eksisterende prosjektet, ville det krevd betydelige endringer for å integrere denne tjenesten. På grunn av tidsbegrensningene i prosjektet, ble det avgjort at en slik implementering ikke ville være gjennomførbar innenfor den gitte tidsrammen.

5.3.2 Coredata Swift

Det var også ønskelig å bruke Core Data i applikasjonen for å håndtere lokale data lagret på enheten. Core Data er et rammeverk fra Apple beskrevet i avsnitt 3.4.2.7. Å bruke Core Data ville forbedret appliaksjonen på en rekke omårder. Det ville blant annet økt ytelsen grunnet rask tilgang til data uten nettverksforespørsler, som igjen ville ført til en bedre brukeropplevelse, særlig ved varierende nettverksforbindelse. Videre gir Core Data også offline tilgjengelighet, slik at brukeren kunne sett og endret data uten nettverkstilkobling for så å synkronisere data når nettverkstilkoblingen ble gjenopprettet. Core Data tilbyr også en rekke verktøy for å definere datamodeller og spørringer som kunne forenklet både utviklingsprosessen og vedlikeholdet av applikasjonen.

5.3.3 E-post passordgjenopprettelse

Dersom en bruker glemmer eller miste innloggingsdetaljene sine bør det være mulig å gjenopprette disse via e-post. En slik tjeneste kan bli implementert via en automatisk e-post som inneholder en lenke for tilbakestilling av passord.

5.3.4 HTTPS forbindelse med SSL/TSL

For å kunne øke sikkerheten på serveren har det kunne blitt implementere SSL/TLS (Secure Sockets Layer/Transport Layer Security)-sertifikater på serveren for å bevise at den er trygg. Disse sertifikatene lar deg benytte HTTPS i stedet for HTTP, noe som sikrer en privat og kryptert forbindelse mellom serveren og brukerne. Den krypterte forbindelsen etableres gjennom en 'TLS handshake' når brukeren besøker appen eller nettstedet, og sikrer at ingen kan fange opp eller manipulere data under utvekslingsprosessen. HTTPS er en nødvendighet når man håndterer sensitiv informasjon som personlige data, betalingsinformasjon eller autentiseringsdetaljer. HTTPS tilbyr brukere en sikker opplevelse ved å beskytte data mot potensielle trusler som hackere og datatyveri. Ved å implementere SSL/TLS-sertifikater opprettholder vi høy sikkerhet og integritet på serveren.

5.3.5 Ferdigstilling og refaktorering av front-end

Ved videre arbeid med applikasjonen vil det være ønskelig å gjennomgå og refaktorere eksisterende kode for å sikre enda bedre etterfølgelse av designprinsipper som MVVM, samt øke cohesion og redusere coupling.

Videre gjenstår det også noe arbeid for at applikasjonen skal være fullstendig. Noen av disse tingene har blitt nevnt tidligere, men det er nødvendig med en grundigere gjennomgang av feilhåndtering. Dette inkluderer utvikling av tester for å sikre funksjonaliteten i front-end.

Testing er essensielt for å sikre at applikasjonen fungerer som forventet. Det bekrefter også at endringer eller nye funksjoner som implementeres ved videreutvikling ikke fører til feil. De viktigste testene som må implementeres er enhetstester og tester av

brukergrensesnittet. På grunn av tidsbegrensninger har det ikke vært mulig å implementere tester i front-end, men dette vil være førsteprioritet ved videre utvikling.

Etter hvert som applikasjonen har blitt utviklet, har det blitt innsett at det finnes bedre løsninger og fremgangsmåter på noen områder, som for eksempel bruk av Core Data. Ved videre utvikling vil det bli brukt enda mer tid på å sette seg inn i konsepter som kan benyttes under utvikling, for så å implementere disse, for å gjøre arbeidet og videreutvikling enklere.

I tillegg til refaktorering og testing er det også viktig å dokumentere koden. Deler av koden er allerede dokumentert, men fullstendig dokumentasjon mangler. Det er lurt å implementere dette da det vil gjøre det enklere å forstå funksjonalitet, og ikke minst vedlikeholde og utvide funksjonaliteten i fremtiden.

Del 6

Konklusjon

Prosjektet har vært svært lærerikt og gitt gruppen verdifull innsikt i moderne teknologi og utviklingsmetodikk. Underveis oppsto flere utfordringer, særlig knyttet til å lære et nytt programmeringsspråk, Swift, innenfor en begrenset tidsramme. Dette krevde betydelig innsats og tilpasning hos gruppen, men bidro samtidig til en bratt læringskurve og bredere forståelse innen utvikling av mobilapplikasjoner og anvendelse av nye teknologier.

En mock-database ble utviklet på grunn av GDPR-restriksjoner knyttet til Solwr's eksisterende databaser, og et baksystem med monolittisk arkitektur ble implementert for å håndtere informasjon om brukere, produkter og leveranser. Utviklingen av dette har gitt gruppen verdifull kunnskap når det kommer til å integrere brukergrensesnitt med et baksystem.

Google Cloud Platform (GCP) ble brukt for hosting av server, noe som sikrer skalerbarhet og pålitelighet. Automatisering av oppdateringer og vedlikehold ble oppnådd gjennom CI/CD-pipelines ved bruk av Terraform og GitHub Actions.

Både prosessen og resultatet har demonstrert for gruppen hvordan moderne teknologier kan skape innovative løsninger som både møter brukernes behov og gjør det mulig å forbedre logistikkprosesser i ulike bransjer i næringslivet.

Gruppen mener selv at dette prosjektet har resultert i en brukervennlig og intuitiv mobilapplikasjon for iOS som forenkler bestillings- og leveringsprosesser, og at den ferdige applikasjonen møter oppdragsgivers kravspesifikasjoner.

Prosjektet har også stort potensial for videreutvikling. Implementeringen av løsningene beskrevet i del 5.3, som inkluderer push-notifikasjoner, e-post for passordtilbakestilling, SSL-tilkobling og integrasjon av Core Data i Swift, kan ytterligere forbedre applikasjonens funksjonalitet og ytelse.

Del 7

Referanser

Referanseliste

[1] Solwr. '*A Solwr solution - Experience Trace*'. In: (). URL: https://solwr.com/products/trace.

[2] Kjetil Sander. 'TCP/IP'. In: (Oct. 2023). URL: https://estudie.no/tcp-ip/.

[3] 'What is Transmission Control Protocol (TCP)?' In: (2024). URL: https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/.

[4] 'TCP 3-Way Handshake Process'. In: (2021).

[5] Nettrafikk.no. 'Hva er HTTPS?' In: (NA). URL: https://nettrafikk.no/seo-tips/hva-er-https.

[6] NTNU. 'rest'. In: (NA). URL: https://folk.ntnu.no/olso/wu/rest/rest.html.

[7] IBM. 'What are Iaas, Paas and Saas?' In: (NA). URL: https://www.ibm.com/topics/iaas-paas-saas.

[8] Google Cloud. 'What is Platform as a Service (PaaS)?' In: (NA). URL: https://cloud.google.com/learn/what-is-paas.

[9] Paessler. 'IT Explained: Server'. In: (). URL: https://www.paessler.com/it-explained/server.

[10] Wikipedia. 'Virtual machine'. In: (). URL: https://en.wikipedia.org/wiki/Virtual_machine.

[11] 'What is a hypervisor?' In: (Jan. 2023). URL: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor.

[12] Victor Elezua. 'What are JWTS?' In: (Sept. 2022). URL: https://goteleport.com/blog/what-are-jwts/.

[13] JWT. 'Introduction to JSON Web Tokens'. In: (). URL: https://jwt.io/introduction.

[14] BvSreyanth. 'Comparison of RS256 and HS256 Algorithms for Token Signing in Cryptography'. In: (2024). URL: https://medium.com/@bvsreyanth/comparison-of-rs256-and-hs256-algorithms-for-token-signing-in-cryptography-bd21e9e7a54d.

[15] Auth0. 'JSON Web Token Structure'. In: (). URL: https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-structure.

[16] Monika Grigutytė. 'What is bcrypt and how does it work?' In: (June 2023). URL: https://nordvpn.com/no/blog/what-is-bcrypt/.

[17] Owasp Cheat Sheet Series. 'Password Storage Cheat Sheet'. In: (). URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[18] Oracle. 'What is a relational database (RDBMS)?' In: (). URL: https://www.oracle.com/database/what-is-a-relational-database/.

[19] Hiren Dhaduk. '10 Software Architecture Patterns You Must Know About'. In: *SimForm* (2020). URL: https://www.simform.com/blog/software-architecture-patterns/.

[20] Estefania Garcia Gallardo. 'What Is MVVM Architecture?' In: *BuiltIn* (2023). URL: https://builtin.com/software-engineering-perspectives/mvvm-architecture.

[21] Sachin Rekhi. 'Don Norman's Principles of Interaction Design'. In: *Medium* (2017). URL: https://medium.com/@sachinrekhi/don-normans-principles-of-interaction-design-51025a2c0f33.

[22] Docker. *Use containers to Build, Share and Run your applications*. https://www.docker.com/resources/what-container/. Tilgangsdato: 02. Mai 2024.

[23] GeeksForGeeks. 'What is CI/CD?' In: (2023). URL: https://www.geeksforgeeks.org/what-is-ci-cd/.

[24] Chandler Harris. 'Microservices vs. monolithic architecture'. In: (). URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[25] Wikipedia. 'Monolithic application'. In: (). URL: https://en.wikipedia.org/wiki/Monolithic_application.

[26] Amazon Web Services. 'What's the Difference Between Monolithic and Microservices Architecture?' In: (). URL: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/.

[27] Grzegorz Blinowski, Anna Ojdowska and Adam Przybyłek. 'Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation'. In: *IEEE Access* 10 (2022), pp. 20357–20374.

[28] Tron Bårdgård. 'Mikrotjenester'. In: (2021). URL: https://ndla.no/nb/subject:26f1cd12-4242-486d-be22-75c3750a52a2/topic:63796e04-10bd-47d7-b3d9-ffc0272e8744/resource:4764c942-f580-49d4-8013-1c2bc258133a.

[29] AlexanderS Gillis. 'Object-oriented programming'. In: *TechTarget* (2021). URL: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP.

[30] GeeksForGeeks. 'Coupling and Cohesion – Software Engineering'. In: (2024). URL: https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/.

[31] Jan-Vidar Tandberg Bakke. 'Uforanderlighet og rene funksjoner — Funksjonell Programmering for Javascript utviklere'. In: *Medium* (2022). URL: https://medium.com/fink-oslo/uforanderlighet-og-rene-funksjoner-funksjonell-programmering-for-javascript-utviklere-91f3f4b409f4.

[32] Amine Benaddi, Jake Shilling and Aristoteles Lopes. 'What is currying in functional programming and how can you use it?' In: *LinkedIn* (2024). URL: https://www.linkedin.com/advice/0/what-currying-functional-programming-how-can-you-bfyhe.

[33] Ståle A Nygård. '*Git på HiNT*'. In: *Høgskolen i Nord-Trøndelag* (2012), pp. 4–9. URL: https://nordopen.nord.no/nord-xmlui/bitstream/handle/11250/146182/Git%20p%C3%A5%20HiNT%20SANygaard.pdf?sequence=4.

[34] Christine Organ and Cassie Bottorf. 'What Are Cross-Functional Teams? Everything You Need To Know'. In: *Forbes* (2022). URL: https://www.forbes.com/advisor/business/cross-functional-teams/.

[35] Alexey Zagalsky et al. '*The Emergence of GitHub as a Collaborative Platform for Education*'. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*. 2015, pp. 1906–1907.

[36] Discord. 'Our Story'. In: (N.A). URL: https://discord.com/company.

[37] Holte Academy. '*Agil eller fossefall - hvilken metode skal vi velge?*' In: (NA). URL: https://www.holteacademy.no/agil-eller-fossefall-hvilken-metode-skal-vi-velge/.

[38] Holte Academy. '*Hva er Scrum og hvilke fordeler gir metodikken?*' In: (NA). URL: https://www.holteacademy.no/hva-er-scrum-og-hvilke-fordeler-gir-metodikken/.

[39] Luc Filion et al. '*Using Atlassian tools for efficient requirements management: An industrial case study*'. In: *2017 Annual IEEE International Systems Conference SysCon*. 2017, pp. 1–2.

[40] Microsoft Azure. 'Hva er Java?' In: (NA). URL: https://azure.microsoft.com/nb-no/resources/cloud-computing-dictionary/what-is-java-programming-language.

[41] Rostislav Fojtik. 'Swift a New Programming Language for Development and Education'. In: Jan. 2020, pp. 284–295. ISBN: 978-3-030-37736-6. DOI: 10.1007/978-3-030-37737-3_26.

[42] Apple. 'Swift - The powerful programming language that's also easy to learn.' In: (NA). URL: https://developer.apple.com/swift/.

[43] Kjell Bratbergsengen, Erlend Tøssebro and Tore Mallaug. 'SQL'. In: (2023). URL: https://snl.no/SQL.

[44] Amazon Web Services. 'What is SQL (Structured Query Language)?' In: (NA). URL: https://aws.amazon.com/what-is/sql/.

[45] ibm. 'What is Java Spring Boot?' In: (). URL: https://www.ibm.com/topics/java-spring-boot.

[46] 'Spring Boot Ecosystem'. In: (Jan. 2024). URL: https://www.geeksforgeeks.org/spring-boot-ecosystem/.

[47] spring.io. 'Spring Security'. In: (). URL: https://spring.io/projects/spring-security.

[48] Javatpoint. 'Lombok Java'. In: (). URL: https://www.javatpoint.com/lombok-java.

[49] John Sundell. 'Discover SwiftUI'. In: *SwiftBySundell* (2023). URL: https://www.swiftbysundell.com/discover/swiftui/.

[50] Apple. 'AVFoundation'. In: *Apple Developer Documentation* (). URL: https://developer.apple.com/documentation/avfoundation/.

[51] Apple. 'Security'. In: *Apple Developer Documentation* (). URL: https://developer.apple.com/documentation/security/.

[52] Apple. 'What Is Core Data?' In: (2018). URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/.

[53] Esat Kemal Ekren. 'What Is Xcode and How to Use It?' In: *netguru* (2024). URL: https://www.netguru.com/blog/what-is-xcode-and-how-to-use-it.

[54] Lukasz Górny. 'Best Java IDEs and Editors Available'. In: *netguru* (2023). URL: https://www.netguru.com/blog/best-java-ides-editors.

[55] Ben Kopf. 'The Power of Figma as a Design Tool'. In: (NA). URL: https://www.toptal.com/designers/ui/figma-design-tool.

[56] ibm. 'What is a REST API?' In: (). URL: https://www.ibm.com/topics/rest-apis.

[57] 'Developing with Spring Boot'. In: (Apr. 2024). URL: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html.

[58] Soumyakanti Ray. 'What is Apache Maven | A Build Automation Tool for Java'. In: (). URL: https://www.turing.com/kb/what-is-apache-maven.

[59] Saurabh Mhatre. 'Understanding Controllers in Spring Boot with Examples'. In: (Nov. 2023).

[60] Databricks. 'Acid transactions'. In: (). URL: https://www.databricks.com/glossary/acid-transactions.

[61] Kjell Bratbergsengen. 'Relasjonsdatabase'. In: (Sept. 2017). URL: https://snl.no/relasjonsdatabase.

[62] Isah Jacob. 'Java Persistence API (JPA) for Database Access'. In: (). URL: https://www.turing.com/kb/jpa-for-database-access.

[63] Dan Arias. In: (Feb. 2021). URL: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/.

[64] William J Buchanan. *BCrypt*. https://asecuritysite.com/hash/bcrypt. Accessed: May 20, 2024. Asecuritysite.com, 2024. URL: https://asecuritysite.com/hash/bcrypt.

[65] Junit. 'he 5th major version of the programmer-friendly testing framework for Java and the JVM'. In: (). URL: https://junit.org/junit5/.

[66] Ishan Gaba. In: (Feb. 2023). URL: https://www.simplilearn.com/tutorials/devops-tutorial/mockito-junit.

[67] docs.spring.io. 'MockMvc'. In: (). URL: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html.

[68] docs.spring.io. 'MockMVC'. In: (). URL: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html.

[69] CodiumAi Team. 'Mock Testing: Understanding the Benefits and Best Practices'. In: (May 2023). URL: https://www.codium.ai/blog/mock-testing/.

[70] Accessibility Guidelines Working Group - Participants. 'Contrast (Minimum) (Level AA)'. In: (N.A). URL: https://www.w3.org/WAI/WCAG21/Understanding/contrast-minimum.html.

[71] Paul Hudson. 'How to create secure text fields using SecureField'. In: *HackingWithSwift* (2022). URL: https://www.hackingwithswift.com/quick-start/swiftui/how-to-create-secure-text-fields-using-securefield.

[72] Apple. 'Picker'. In: (N.A). URL: https://developer.apple.com/documentation/swiftui/picker.

[73] Apple. 'Supporting Dark Mode in Your Interface'. In: (N.A). URL: https://developer.apple.com/documentation/uikit/appearance_customization/supporting_dark_mode_in_your_interface.

[74] Apple. 'Storing Keys in the Keychain'. In: (N.A). URL: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_keychain.

[75] Paul Hudson. 'Actors'. In: (N.A). URL: https://www.hackingwithswift.com/swift/5.5/actors.

[76] Flutter. 'Build for any screen'. In: (N.A). URL: https://flutter.dev.

[77] 'What is web socket and how it is different from the HTTP?' In: (Apr. 2023). URL: https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/.

[78] 'What are the benefits of Kubernetes?' In: (). URL: https://cloud.google.com/learn/what-is-kubernetes.

[79] Jonathon Albert. 'Implement Firebase Push Notifications in Swift'. In: (2022). URL: https://betterprogramming.pub/implement-firebase-push-notifications-in-swift-1c5f4460a28.

Del A

Vedlegg

Appendix

A Vedlegg - Forprosjektplan

Se vedlagt mappe: Vedlegg/Vedlegg-A-Forprosjektplan.pdf

B Vedlegg - Prosjekthåndbok

Se vedlagt mappe: Vedlegg/Vedlegg-B-prosjekthåndbok

C Vedlegg - KI Deklarasjon

Se vedlagt mappe: Vedlegg/Vedlegg-C-KI-deklarasjon.pdf

D Vedlegg - Postman tester

Se vedlagt mappe: Vedlegg/Vedlegg-D-Postman-tester

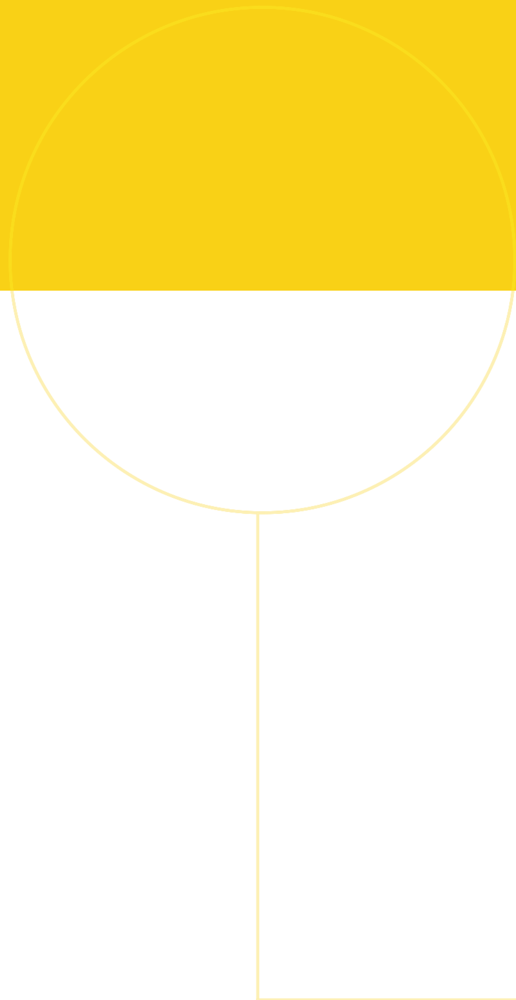
E Vedlegg - Kildekode

Se vedlagt mappe: Vedlegg/Vedlegg-E-Kildekode

Lenke til GitHub repository:

Frontend: https://github.com/IDATA2900-Bacheloroppgave/Front-end

Backend: https://github.com/IDATA2900-Bacheloroppgave/WarehouseManagementSystem



NTNU

Kunnskap for en bedre verden