

Fleet Insight



Smartere beslutninger på tværs af flåden.

Aarhus Universitet

2025E74

Vejleder: Henrik Daniel Kjeldsen

Zaki Nahimi, Jonas *202109834*

Göktas, Deniz *202207803*

Ipek, Mikail *202206885*

Dato: *12. December 2025*

Anslag (ekskl. figurer, tabeller, fotos): *72.063*

1 Resume

Denne bachelor omhandler udviklingen af et flådestyringssystem, der giver virksomheder et samlet og realtidsopdateret overblik over deres køretøjer. Til dette formål er der udviklet en simulator, et Web API og en webapplikation, som tilsammen udgør Fleet Insight-systemet. Simulatoren genererer telemetridata og sender dem via MQTT til API'et, hvor data behandles, lagres og distribueres videre til klienten. Herfra kan brugerne følge køretøjers positioner, ruter og status samt kommunikere gennem indbygget chat.

For at sikre korrekt adgang og håndtering af følsomme oplysninger er der implementeret rollebaseret login. Systemet understøtter både realtidskommunikation og alarmsystemer, der hjælper koordinatorer med at reagere hurtigt på relevante hændelser.

2 Abstract

This bachelor project concerns the development of a fleet management system that provides companies with a unified, real-time overview of their vehicles. For this purpose, a simulator, a Web API, and a web application have been developed, together forming the Fleet Insight system. The simulator generates telemetry data and publishes it via MQTT to the API, where the data is processed, stored, and forwarded to the client. From the web interface, users can monitor vehicle positions, routes, and status, as well as communicate through a built-in chat feature.

To ensure secure access and protection of sensitive information, role-based authentication has been implemented. The system supports real-time communication and an alert mechanism that assists coordinators in responding quickly to important events.

Indhold

1	Resume	1
2	Abstract	1
3	Forord	5
3.1	Ordliste	6
4	Indledning	7
4.1	Problem beskrivelse	7
4.1.1	Problemformulering	7
4.2	Vores produkt	7
4.3	Ansvarsområder	8
5	Kravspecifikation	11
5.1	Ikke-funktionelle krav	20
5.2	Afgrænsning	21
5.3	MoSCoW	21
5.3.1	Funktionelle krav	21
5.3.2	Ikke-funktionelle krav	22
6	Metode og Proces	23
6.1	Metode	23
6.1.1	Softaredokumentation	23
6.2	Proces	23
6.2.1	Gruppedannelse	23
6.2.2	Processamarbejde	23
6.2.3	Planlægning og møder	24
6.2.4	Agilitet	24
6.2.5	Azure DevOps og GitHub	24
6.3	Versionskontrol – Git	25
7	Analyse	27
7.1	Nuværende muligheder	27
7.2	Integration og begrænsninger	27
8	Softwarearkitektur	28
8.1	C4 og systemanalyse	28
8.2	C4 model afgrænsning	28

8.2.1	C1 diagram	28
8.2.2	C2 diagram	29
8.3	Systemsekvensdiagram	30
8.3.1	Opret bruger	30
8.3.2	Flåde	32
8.3.3	Chat	33
8.4	CI pipeline og Build-proces	34
8.5	Valg af teknologi	34
8.5.1	Simulator teknologianalyse	34
8.5.2	Web API teknologianalyse	36
8.5.3	Webapp teknologianalyse	37
9	Software Design	38
9.1	Domæne model	38
9.2	Simulator design	39
9.3	Web API design	41
9.4	Webapp design	43
10	Software implementering	46
10.1	Simulator implementering	46
10.1.1	Simulator folder struktur	46
10.1.2	Generering af køreprofil	47
10.1.3	Periodisk udsendelse af telemetri	48
10.1.4	Manuelle ruteændringer via MQTT	50
10.2	Web API implementering	51
10.2.1	Web API folder struktur	52
10.2.2	Live telemetri	53
10.2.3	Rute og køretøjsopdateringer	54
10.2.4	Chat 1:1	56
10.2.5	Authentication implementation	57
10.3	Webapp implementering	58
10.3.1	Resultatet af udviklingen	58
10.3.2	Webapp folder struktur	60
10.3.3	Hentning og normalisering af køretøjsdata	61
10.3.4	Manuel ruteændring fra kortet	62
10.4	Infrastruktur og deployment	63
10.4.1	Hosting	64
10.4.2	CI pipeline	64
11	Test	66

11.1	Test af simulatoren	66
11.1.1	Unit tests	66
11.1.2	Smoke-test	67
11.2	Test af Web API	67
11.2.1	Unit test	67
11.2.2	Integrationstest	68
11.2.3	Smoke-test	68
11.3	Test af webapp	69
11.3.1	Unit tests	69
11.4	Integration mellem webapp og Web API	70
12	Accepttest	71
12.1	Resultater af accepttest	71
13	Diskussion	76
13.1	Resultater	76
13.2	Styrker og begrænsninger	76
13.3	Refleksion over teknologivalg	77
13.4	Perspektivering	77
14	Konklusion	78
15	Fremtidigt arbejde	79
15.1	Integration med fysisk hardware	79
15.2	Udvidet funktionalitet	79
15.3	Sikkerhed og databeskyttelse	79
15.4	Brugerinddragelse og pilottest	80
16	Appendiks	81
Referencer		83

3 Forord

Denne bachelor er udarbejdet af Mikail Ipek, studienummer 202206855, Deniz Göktas, studienummer 202207803, og Jonas Zaki Nahimi, studienummer 202109834, som er 7. semesters Softwareteknologi-studerende på Aarhus Universitet, Institut for Elektro og Computerteknolog. Bachelorprojektet er udarbejdet internt i gruppen og under vejledning af Henrik Daniel Kjeldsen.

Bachelorrapporten afleveres den 12. Decemeber 2025 og forsvarer mundtligt den 23/01/2026, hvor bedømmelsen også vil finde sted.

3.1 Ordliste

Forkortelse	Begreb
OTP	One-time-password
ETA	Estimated Time of Arrival
OSRM	Open Source Routing Machine
MQTT	Message Queuing Telemetry Transport
DTO	Data Transfer Object
PR	Pull Request
CI	Continous Integration
MVP	Minimum Viable Product
QoS	Quality Of Service

Tabel 3.1: *Ordlistetabel*

4 Indledning

4.1 Problem beskrivelse

Flådestyring i transport- og logistikbranchen afhænger i stigende grad af reeltidsdata, som kan give overblik over køretøjers position, ruter, fremdrift og eventuelle afvigelser. Flere faglige udgivelser peger på, at reeltidsdata kan forbedre både effektivitet, sikkerhed og beslutningsgrundlag i daglig drift [1]–[4]. Disse løsninger fremhæver især behovet for synlighed på tværs af køretøjer, mulighed for hurtige reaktioner ved hændelser og en samlet adgang til telemetri, ruter og køretøjsstatus.

Eksisterende systemer tilbyder typisk enkelte af disse elementer, men mangler ofte en samlet, konsistent platform hvor flådemedlemmer — såsom administratorer, koordinatorer og chauffører — kan dele informationer i realtid og tilgå relevante værktøjer samme sted. Moderne systemer kombinerer ofte positionering, rutehåndtering, alarmer og kommunikation, men der er fortsat et behov for at undersøge, hvordan en integreret løsning kan gøre dette mere overskueligt og operationelt for brugerne [2], [3].

Med afsæt i denne problemstilling undersøger dette projekt, hvordan reeltidsdata, telemetri og rutehåndtering kan samles i ét værktøj, der understøtter både drift, overblik og kommunikation i en flåde.

4.1.1 Problemformulering

Hvordan kan man give flådekoordinatorer og chauffører et samlet, reeltidsopdateret overblik over køretøjsstatus, ruter og beskeder, så de kan træffe hurtige beslutninger og reagere rettidigt på hændelser?

4.2 Vores produkt

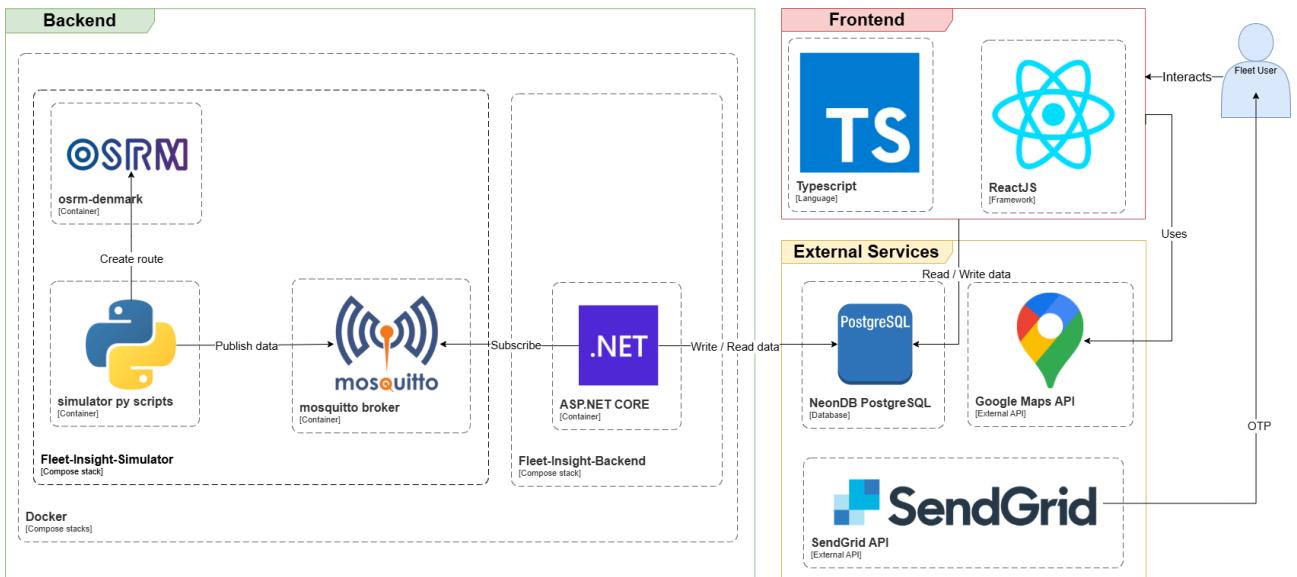
Fleet Insight er et flådestyringssystem bestående af en React-webapp og et ASP.NET Core Web API. Webappen giver brugerne et live Google Maps-kort med køretøjspositioner, listevisning med data, CSV-eksport, automatisk genererede advarsler (lavt brændstof, offline, høj hastighed), mulighed for manuel ruteændring samt 1:1-chat. Alle visninger opdateres i realtid via SignalR, så både chauffør og koordinator altid ser den samme aktuelle status.

Web API'et håndterer sikker login, rolleadministration og OTP-udsendelse via SendGrid, kommunikerer med en PostgreSQL-database, modtager simuleret telemetri via MQTT og

udsender liveopdateringer gennem dedikerede SignalR-hubs for både køretøjsdata og chat. Ruteændringer publiceres som MQTT-beskeder til køretøjerne eller simulatoren. Simulatoren anvender OSRM til at beregne ruter og ETA og udsender realistiske telemetridata via MQTT, så hele reeltidsflowet kan afprøves uden fysiske køretøjer.

Tilsammen giver første version af systemet ét samlet sted, hvor flådekoordinatorer kan overvåge køretøjer, følge ruter, eksportere data og kommunikere effektivt med chauffører i realtid.

På nedenstående figur 4.1 kan systemet ses skitseret.



Figur 4.1: Systemoversigt: Python-simulator og OSRM genererer ruter og publicerer telemetri via Mosquitto; .NET-Web API'et subscriber, persisterer i PostgreSQL og sender/afsender via Google Maps og SendGrid; webapp i React/TypeScript bruger Web API'et, mens brugeren interagerer med UI'et.

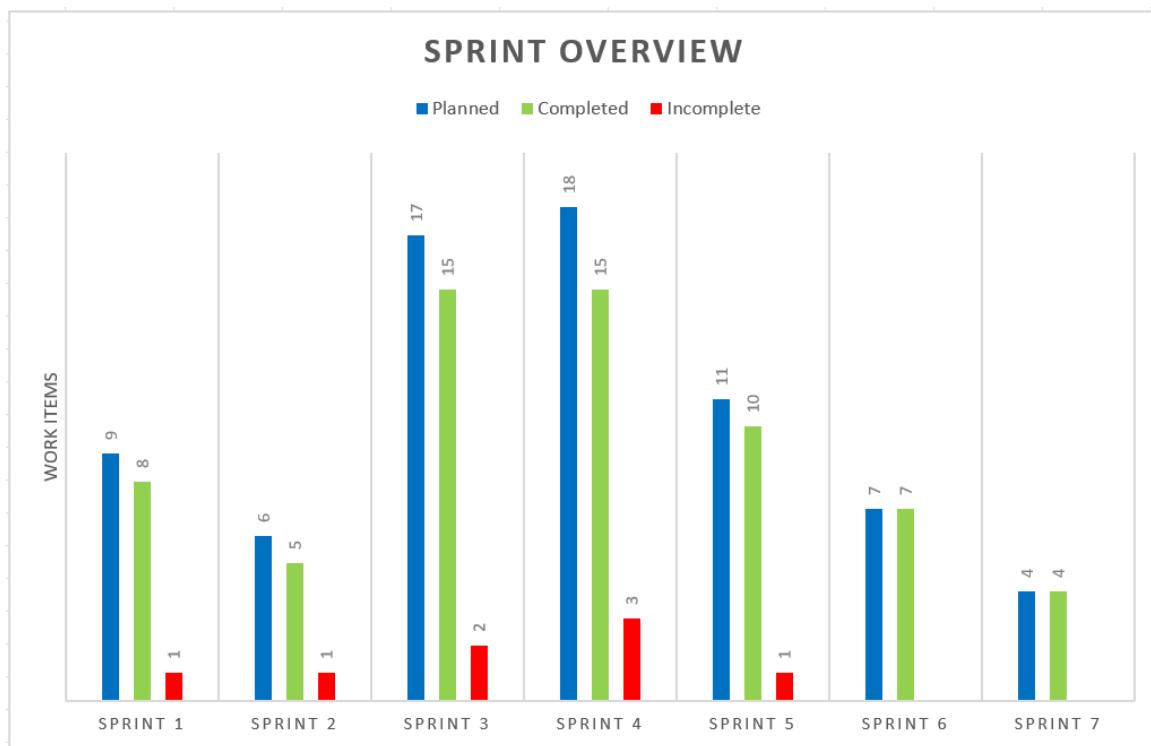
4.3 Ansvarsområder

Tabel 4.1 giver et samlet overblik over, hvilke dele af bachelorprojektet de forskellige gruppemedlemmer har haft hovedansvaret for.

Område	Deniz	Mikail	Jonas
Web API	P	S	P
Webapp	P	P	
Simulator		P	S
Rapport	P	P	P

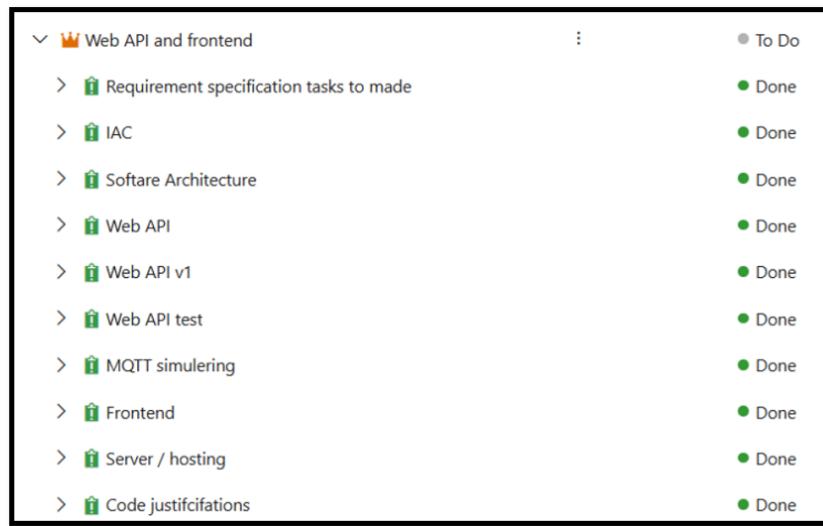
Tabel 4.1: Oversigt over ansvarsområder i projektet, hvor P markerer den person, der har haft hovedansvaret, og S indikerer den, der har haft en sekundær rolle.

Figur 4.2 viser en samlet oversigt over arbejdsfordelingen og fremdriften på tværs af projektets sprints. For hvert sprint præsenteres tre kategorier af work items: Planned, Completed og Incomplete, hvilket giver et tydeligt billede af både planlagt arbejdsmfang og det faktiske output.



Figur 4.2: Diagrammet giver et samlet overblik over projektets sprints. De blå søjler viser antal planlagte opgaver, de grønne viser de opgaver der blev gennemført, og de røde repræsenterer de opgaver, som ikke blev færdiggjort i det pågældende sprint. Generelt ligger completed tæt på det planlagte i de fleste sprints, særligt i Sprint 3, 4 og 5, mens antallet af incomplete opgaver gennemgående er lavt. Sprint 6 og 7 indeholder færre opgaver, hvilket afspejler en mere afsluttende fase af projektet.

Figur 4.3 viser et overblik over de opgaver, der er blevet gennemført i projektet på tværs af Web API’et, webappen og MQTT-simulering. Figuren giver et samlet indblik i den funktionelle backlog, som er behandlet i udviklingsforløbet. For en fuld gennemgang af backloggen henvises der til bilag 6.c.



Figur 4.3: Overblik over de opgaver, der er blevet fuldført i projektet, herunder udviklingen af Web API, webapp, test, simulator og øvrige relaterede dele af systemet.

5 Kravspecifikation

Fleet Insight er udviklet i en agil udviklingsproces, hvor krav og funktionalitet er blevet udfoldet gradvist. Hvert sprint har givet anledning til nye indsigter og krav, som efterfølgende er blevet håndteret i den videre udvikling. Denne iterative tilgang har gjort det muligt at reagere på ændringer undervejs og samtidig sikre, at systemet adresserer de centrale behov.

I begyndelsen af projektet blev der fastlagt nogle overordnede funktionelle og ikke-funktionelle krav, som beskrev den primære funktionalitet af Fleet Insight. Disse krav har dannet grundlag for arkitektur og design og er blevet justeret gennem udviklingsforløbet. Fokus har været på at skabe en platform, hvor virksomheder kan få et samlet og overskueligt real-time overblik over deres køretøjer og administrere flådedata.

User story 1:

Som bruger ønsker jeg at kunne logge ind, så jeg kan få adgang til systemet.
Accepttest: Egenskab: Login Brugeren skal logge ind for at få adgang til dashboardet.
Baggrund: Givet at en bruger befinner sig på login-siden
Scenarie: Gyldigt login Når brugeren indtaster gyldige loginoplysninger Så skal brugeren føres til systemet Og sessionen skal være aktiv, så der ikke kræves nyt login ved sideskift
Scenarie: Ugyldig login Når brugeren indtaster forkerte loginoplysninger Så skal systemet afvise forsøget Og brugeren skal se en fejlmeldelse
Scenarie: Glemt login Når brugeren har glemt adgangskoden Så kan han trykke på "glemt adgangskode" Og indtaste hans mail og trykke på "Send nulstilling-link" Og få tilsendt en mail med en engangskode Så kan han trykke på "engangskode" for at navigere til engangskode siden Og blive bedt om indtaste en ny adgangskode
Scenarie: Brugerens første login Når brugeren logger ind for første gang Så skal brugeren trykke på "engangskode" for at navigere til engangskode siden Og indtaste engangskoden Så skal systemet tvinge medarbejderen til at ændre engangskoden

Tabel 5.1: User story 1 - Log ind

User story 2:

Som admin ønsker jeg at kunne oprette nye brugere, så nye brugeren kan få adgang til systemet
Accepttest: Egenskab: Administration Systemet skal understøtte oprettelse af brugere, så medarbejdere kan få adgang.
Baggrund: Givet at en admin er logget ind
Scenarie: Opret bruger med gyldige oplysninger Når admin indtaster e-mail, navn, alder og rolle Og klikker "opret bruger" Så skal systemet oprette brugeren og sende en bekræftelsesmail
Scenarie: Opret bruger med manglende oplysninger Når admin forsøger at oprette en bruger uden at udfylde obligatoriske felter Så skal systemet afvise oprettelsen af brugeren Og admin skal se fejlmeldelsen "Dette felt er påkrævet"

Tabel 5.2: *User story 2 - Opret bruger*

User story 3:

Som bruger med adgang ønsker jeg at kunne se køretøjer live på et kort med tilhørende data, så jeg kan få overblik.
Accepttest: Egenskab: Live overblik Systemet skal give brugere med adgang et live overblik over flåden.
Baggrund: Givet at en bruger er logget ind Og befinder sig på siden "Flåde"
Scenarie: Se live køretøjsdata Når brugeren åbner flådesiden Så skal systemet vise et kort med alle køretøjer, samt en køretøjsliste med relevant telemetri data Og positioner og data skal opdateres løbende uden at siden genindlæses.
Scenarie: Live-data kan ikke hentes Når brugeren åbner flåde Men HTTP-kaldet til køretøjer fejler eller SignalR-forbindelsen til vehicleHub ikke kan oprettes Så vises en fejlbesked om manglende live-data Og SignalR forsøger at genoprette forbindelsen automatisk.
Scenarie: Kortet kan ikke indlæses Når brugeren åbner flåde Men Google Maps API-nøglen mangler eller Maps SDK-loaderen fejler Så vises en fejlbesked om, at kortet ikke er tilgængeligt Og <u>køretøjs</u> listen vises stadig, så data kan ses.
Scenarie: Rediger rute Når brugeren klikker på knappen "Fokus" på et specifik køretøj

<p>Så kan brugeren klikke på knappen "Rediger rute"</p> <p>Og have mulighed for at ændre destinationen ved indtaste den nye destination i det redigerbar felt</p> <p>Og trykke på "Opslag" knappen</p> <p>Og derefter klikke på knappen "Send rute" for at opdatere ruten</p> <p>Og systemet skal meddele den tilknyttede driver om den opdaterede rute med en notifikation</p> <p>Så driveren kan se den opdaterede rute</p>
<p>Scenarie: Ruteopdatering mislykkes</p> <p>Når brugeren sender en rute-opdatering</p> <p>Og Web API'et afviser anmodningen eller der ikke kan oprettes forbindelse til MQTT-broker</p> <p>Så vises en fejlbesked om, at ruten ikke blev sendt</p> <p>Og brugeren kan rette input eller prøve igen</p>
<p>Scenarie: Dataeksport</p> <p>Når brugeren vil eksportere data</p> <p>Så kan brugeren vælge mellem prædefinerede datoer eller indtaste eget interval</p> <p>Og kan vælge mellem prædefinerede statusser</p> <p>Og hente dataen ved at klikke på knappen "Hent CSV"</p>
<p>Scenarie: CSV-eksport fejler</p> <p>Når brugeren forsøger at hente CSV Men eksport kaldet fejler (fx 401 eller serverfejl)</p> <p>Så vises en fejlbesked om, at eksporten ikke lykkedes</p>
<p>Scenarie: Koordinering af alarmer</p> <p>Når en driver kører for stærkt, har for lavt brændstof eller driverens køretøj ikke har afsendt data i fem minutter</p> <p>Så modtager koordinatoren en eller flere alarmer baseret på hændelsen</p> <p>Og kan se samt lukke dem.</p>

Tabel 5.3: *User story 3 – Flåde*

User story 4:

Som bruger ønsker jeg at kunne logge ud, Så min session afsluttes og mine data beskyttes.
Accepttest: Egenskab: Afslut session Systemet skal give brugere mulighed for at logge ud, så deres session afsluttes og data forbliver beskyttet.
Baggrund: Givet at en bruger er logget ind
Scenarie: Veludført log ud Når brugeren trykker på knappen "Log ud" Så skal systemet afslutte sessionen Og brugeren skal føres tilbage til login siden Og forsøg på at åbne beskyttede sider skal kræve et nyt login
Scenarie: Automatisk log ud ved timeout Når en bruger har været inaktiv i en pre-defineret periode på ti minutter Så skal systemet automatisk afslutte sessionen Og brugeren skal føres til login siden

Tabel 5.4: *User story 4 - Log ud*

User story 5:

Som bruger Ønsker jeg at kunne chatte 1:1 med en kollega Så vi hurtigt kan opdatere hinanden om drift og hændelser.
Accepttest: Egenskab: Kommunikation Systemet skal give brugere mulighed for at kommunikere med kollegaer.
Baggrund: Givet at en bruger er logget ind og vil kommunikere med en kollega
Scenarie: Veludført kommunikation Når en bruger trykker på knappen "Chat" åbnes chat-modalen Og klikker på en kollega fra listen Så indlæser systemet de seneste beskeder i tråden Når brugeren skriver en besked og trykker på knappen "Send" Så vises beskeden i brugerens chat med status "Sendt" Og kollegaen får en notifikation, samt modtager beskeden i sin chat
Scenarie: Fejlet kommunikation Når bruger åbner chatten for en kollega Og skriver en besked og trykker "Send" Så bliver beskeden ikke sendt Og systemet viser fejlbeskeden "Kunne ikke sende beskeden. Prøv igen." Og der oprettes ingen ny besked i databasen

Tabel 5.5: User story 5 - Chat

User story 6:

Som admin ønsker jeg at kunne ændre eksisterende brugeres roller, så adgangsstyring altid er korrekt og opdateret.
Accepttest: Egenskab: Rolleændring Systemet skal give en admin mulighed for at ændre eksisterende brugeres roller.
Baggrund: Givet at en admin er logget ind Og befinder sig på siden "Administration"
Scenarie: Ændre rolle på eksisterende bruger Givet at brugeren "maria@firma.dk" har rollen "Driver" Når admin vælger en ny rolle fra listen af roller Så skal systemet opdatere brugerens nye rolle til opdaterede rolle
Scenarie: Fejl ved ændring af rolle Givet at brugeren "maria@firma.dk" har rollen "Driver" Når admin vælger en ny rolle fra listen af roller Og systemet oplever en fejl i databasen Så skal systemet afvise ændringen Og vise beskeden "Rollen kunne ikke opdateres"

Tabel 5.6: User story 6 - Ændring af roller

User story 7:

Som bruger ønsker jeg at kunne se og redigere min profil så mine oplysninger altid er opdaterede og korrekte.
Accepttest: Egenskab: Profil Systemet skal give brugeren mulighed for at se og redigere sin profil.
Baggrund: Givet at en bruger er logget ind Og befinder sig på siden "Profil"
Scenarie: Se profiloplysninger Når brugeren åbner profilsiden Så skal systemet vise profilbilledet Og vise navnet og andre registrerede oplysninger
Scenarie: Opdater profil med gyldige oplysninger Når brugeren ændrer sit navn til "Jakob Hansen" Og trykker på knappen "Gem ændringer" Og uploader et nyt profilbillede Så skal systemet opdatere profiloplysningerne Og der vises en toast på UI'en
Scenarie: Opdater profil med manglende oplysninger Når brugeren forsøger at gemme uden at udfylde de obligatoriske felter Så skal systemet afvise ændringen Og vise en fejmeddelelse "Dette felt er påkrævet"
Scenarie: Fejl ved opdatering af profil Når brugeren ændrer oplysninger og trykker "Gem ændringer" Og systemet oplever en serverfejl Så skal systemet afvise ændringen Og vise beskedten "Failed to fetch"

5.1 Ikke-funktionelle krav

- IF.1 Systemet understøtter sikkert login med rollebaseret adgang for Driver, Koordinator og Admin.
- IF.2 Systemet viser køretøjer på et live-kort, hvor positioner opdateres automatisk, og hvor bruger kan søge og filtrere på nummerplade, rute og mærke.
- IF.3 Systemet giver mulighed for 1:1-chat med sendt-/læst-status, håndtering af fejl ved mislykket afsendelse samt notifikationer ved modtagede beskeder.
- IF.4 Systemet viser brugervenlige fejlmeddelelser i tilfælde af API-fejl eller nedbrud i realtidsforbindelsen.
- IF.5 Systemet loader flådesiden og live-kortet på 2 sekunder.
- IF.6 Systemet håndterer og lagre brugerdata sikkert i PostgreSQL, herunder hashed adgangskoder.
- IF.7 Systemet opdaterer chaufførens rute inden for 10 sekunder efter, at en admin eller koordinator har sendt en manuel ruteændring – uden at kræve refresh.
- IF.8 Systemet giver mulighed for at downloade telemetridata i CSV-format.
- IF.9 Systemet giver bruger mulighed for at opdatere navn, e-mail og adgangskode med validering og meningsfulde fejlmeddelelser.
- IF.10 Systemet afvikler en testpipeline, der automatisk bygger projektet og kører relevante tests.
- IF.11 Systemet logger automatisk bruger ud efter 10 minutters inaktivitet og kræver login igen.
- IF.12 Systemet genererer alarmer for lavt brændstof, offline-status og høj hastighed, og dismissal skal persisteres pr. bruger.
- IF.13 Systemet understøtter engelsk som ekstra sprog i webappen.
- IF.14 Systemet giver mulighed for at sortere køretøjer efter hastighed, seneste telemetri eller brændstofniveau samt filtrere efter fx brændstof $\leq 20\%$.
- IF.15 Systemet opnår en test coverage på over 80%.
- IF.16 Systemet understøtter en multitenant arkitektur.
- IF.17 Systemet muliggør, at bruger klikker på et køretøj og sender en “stop”-kommando.

IF.18 Systemet inkluderer en fuld analytics- eller statistikside.

IF.19 Systemet understøtter skalering til 200 køretøjer eller 10+ samtidige brugere, medmindre dette måles særskilt.

5.2 Afgrænsning

Eftersom dette er et bachelorprojekt, afgrænses *Fleet-Insight* således, at en sammenhængende og demonstrerbar løsning kan bygges, uden at strække projektets rammer.

Der lægges ikke vægt på integration med fysisk køretøjs-hardware. I stedet simuleres telemetri (GPS, hastighed, brændstof mm.) softwaremæssigt, og systemet udformes, så reel integration vil let-gøres i fremtidigt arbejde.

5.3 MoSCoW

I forbindelse med afgrænsningen af *Fleet Insight* er der udarbejdet en MoSCoW-analyse baseret på de krav, der er defineret gennem projektets user stories. Da der løbende er opstået nye krav og ændringer som følge af designiterationer og arkitekturvalg, er analysen blevet opdateret undervejs. Der er således gjort brug af en iterativ MoSCoW-analyse, hvor hvert krav er blevet prioriteret som et skal (**must**), bør (**should**), kan (**could**) eller vil ikke (**won't**) krav.

Nedenfor præsenteres *Fleet-Insight's* MoSCoW prioritering, der er kategoriseret som **must**-krav. For at se hele afgrænsningen af de resterende krav til systemet, se bilag [1.b](#)

5.3.1 Funktionelle krav

Alle user stories er blevet prioriteret som **must** krav for systemet.

M.1 Log ind

M.2 Opret Bruger

M.3 Flåde

M.4 Log ud

M.5 Chat

M.6 Ændring af Roller

M.7 Profil

5.3.2 Ikke-funktionelle krav

- M.8 Systemet skal understøtte sikkert login med rollebaseret adgang for Driver, Koordinator og Admin.
- M.9 Systemet skal kunne vise køretøjer på et live-kort, hvor positioner opdateres automatisk, og hvor brugeren kan søge og filtrere på nummerplade, rute og mærke.
- M.10 Systemet skal give mulighed for 1:1-chat med sendt-/læst-status, håndtering af fejl ved mislykket afsendelse samt notifikationer ved modtagede beskeder.
- M.11 Systemet skal vise brugervenlige fejlmeddelelser i tilfælde af API-fejl eller nedbrud i realtidsforbindelsen.
- M.12 Systemet skal loade flådesiden og live-kortet på 2 sekunder.
- M.13 Systemet skal håndtere og lagre brugerdata sikkert i PostgreSQL, herunder hashed adgangskoder.
- M.14 Systemet skal opdatere chaufførens rute inden for 10 sekunder efter, at en admin eller koordinator har sendt en manuel ruteændring – uden at kræve refresh.
- M.15 Systemet skal give mulighed for at downloade telemetridata i CSV-format.
- M.16 Systemet skal give brugeren mulighed for at opdatere navn, e-mail og adgangskode med validering og meningsfulde fejlmeddelelser.

6 Metode og Proces

6.1 Metode

Dette afsnit vil beskrive de metoder, der er anvendt. For at se flere informationer om metode, se bilag [6.b](#).

6.1.1 Softaredokumentation

For at dokumentere softwarearkitekturen er der taget udgangspunkt i C4-modellen. C4-modellen er valgt, da den giver en struktureret og overskuelig opdeling af systemet samtidig med, at den understøtter en agil udviklingsproces. Mere traditionelle metoder til arkitekturdokumentation lægger ofte op til en ikke-agil tilgang, hvor hele systemet beskrives og visualiseres tidligt i forløbet. Dette kan medføre omfattende dokumentation, som kræver løbende revision og opdatering, efterhånden som arkitekturen udvikler sig.

Selvom store dele af systemets arkitektur forventes at være stabile gennem udviklingsforløbet, vil enkelte komponenter naturligt ændre sig i takt med iterationer og nye krav. For at supplere C4-modellen udarbejdes der sekvensdiagrammer, som illustrerer systemets flow og giver et mere detaljeret indblik i, hvordan brugeren interagerer med systemet.

6.2 Proces

Dette afsnit beskriver den overordnede proces bag udviklingen af *Fleet Insight*. For en mere detaljeret gennemgang af processen henvises der til bilag [6.b](#).

6.2.1 Gruppedannelse

Bachelorgruppen blev dannet på baggrund af tidlige samarbejde med en velfungerende dynamik mellem gruppens medlemmer. I projektets indledende fase blev ansvarsområderne fordelt ud fra de enkelte medlemmers kompetencer og interesser. Mikail havde hovedansvaret for udviklingen af systemets simulator, mens Deniz og Jonas var ansvarlige for Web API-udviklingen. Webapp-udviklingen blev hovedsageligt implementeret af Mikail og Deniz.

6.2.2 Processamarbejde

Eftersom gruppen kun består af tre personer, blev det besluttet ikke at udarbejde en formel samarbejdskontrakt. Denne beslutning udsprang af, at gruppemedlemmerne allerede kender

hinanden indgående fra tidligere semestre, hvor der stort set er blevet fulgt de samme kurser og arbejdet sammen i forskellige projekter.

Det tidligere samarbejde og kendskabet til hinandens arbejdsstil skabte en høj grad af tillid omkring møder, aftaler og opgavehåndtering. Derfor blev en uformel og tillidsbaseret arbejdsproces vurderet som både tilstrækkelig og mest effektiv for projektet.

6.2.3 Planlægning og møder

Under den indledende samtale blev gruppens arbejdstider fastlagt, og der blev udarbejdet et skema (bilag 6.d) med angivelse af, hvornår der kunne arbejdes på bachelorprojektet. Det blev planlagt, at der primært skulle arbejdes mandag, tirsdag og onsdag, med mulighed for at inddrage øvrige dage efter behov.

Vejledermøder blev lagt om mandagen eller tirsdagen. I projektets start blev både sprint planning og retrospektiv afholdt om fredagen, hvilket passede bedst i forhold til vejledermøder og gruppens samlede arbejdstid. Sprint planning og retrospektiver blev gennemført hver anden uge.

6.2.4 Agilitet

Projektet fulgte en agil arbejdsmetode med iterative sprints af to ugers varighed. Dette gjorde det muligt løbende at evaluere fremskridt og hurtigt tilpasse udviklingen på baggrund af nye erfaringer.

Efter hver iteration blev der udført evaluering og test, hvilket dannede grundlag for at identificere nye krav og forbedringspunkter, som derefter blev indarbejdet i næste sprint. Denne tilgang gav en fleksibel udviklingsproces, reducerede behovet for større omarbejdninger og styrkede kvaliteten af den endelige løsning, da produktet løbende blev justeret efter projektets mål og brugernes behov.

6.2.5 Azure DevOps og GitHub

I projektet blev både Azure DevOps og GitHub anvendt som centrale udviklingsværktøjer. Azure DevOps blev brugt til planlægning og koordinering via Boards, hvor sprints og opgaver blev oprettet, prioriteret og fulgt løbende. Dette gav et klart overblik over fremdrift og arbejdsfordeling.

GitHub fungerede som hosting-platform for projektets Git-repositories samt som samarbejds-værktøj gennem branches og PR's. Denne opsætning understøttede versionsstyring, kodegen-

nemgang og en struktureret håndtering af ændringer.

6.3 Versionskontrol – Git

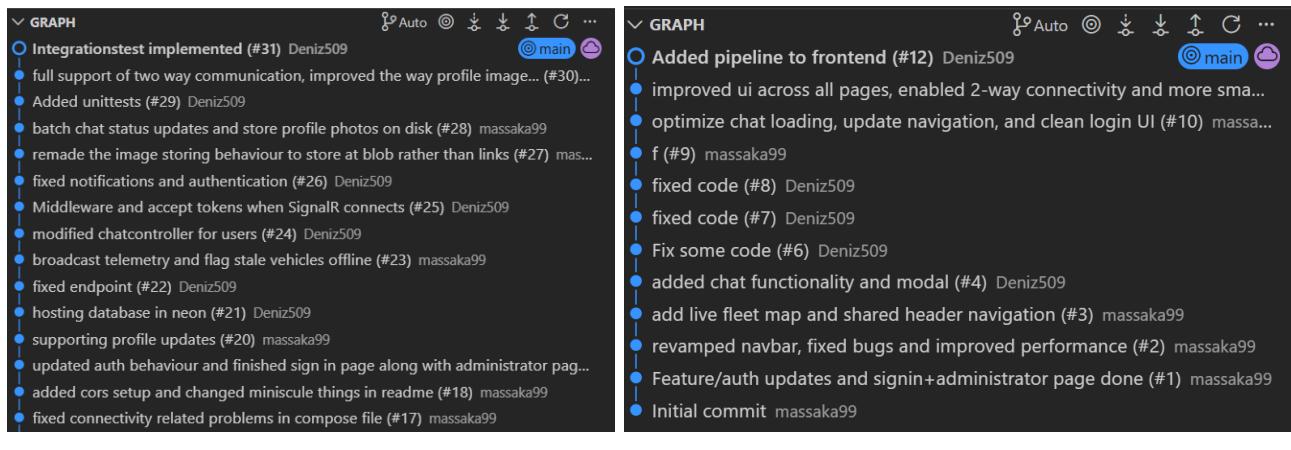
Git blev anvendt som versionsstyringsværktøj for projektets kodebase, organiseret i tre repositories: Web API’et, simulatoren og webappen.

Udviklingen fulgte en feature-branching-strategi, hvor *main*-branchen repræsenterede den stabile og deployable del af systemet. Nye funktioner blev udviklet i separate feature-branches, så arbejdet kunne foregå parallelt uden at påvirke den stabile kode.

Når en funktionalitet var færdigudviklet, blev der oprettet et PR for at sammenflette ændringerne med *main*. PR’s blev brugt til review, test og konflikthåndtering samt til at sikre kvalitet i ændringerne. Først efter godkendelse blev ændringerne merged, hvilket sikrede en kontrolleret og stabil udviklingsproces.

Denne arbejdsform understøttede høj kodekvalitet og effektivt samarbejde samt minimerede risikoen for fejl i den stabile kodebase.

På figur 6.1 og figur 6.2 ses henholdsvis commit- og PR-historikken for Web API’et og webappen. Af figurerne fremgår brugen af feature branching gennem de gentagne merges ind i *main*-branchen.



Figur 6.1: Commit-historikken for Web API og webapp viser, at udviklingen primært er sket på feature branches, som derefter er flettet ind i *main* via squash merges. Dermed samles ændringerne til én merge-commit per feature branch.

Author	Label	Projects	Milestones	Reviews	Assignee	Sort
0 Open	✓ 31 Closed					
Integrationtest implemented ✓						
#1 by Denzel09 was merged 2 days ago • Approved						
full support of two way communication, improved the way profile image... ✓						
#30 by masokuk99 was merged 2 days ago • Approved						
Added unittests ✓						
#4 by Denzel09 was merged 3 weeks ago • Approved						
batch chat status updates and store profile photos on disk ✓						
#39 by masokuk99 was merged last month • Approved						
remade the image storing behaviour to store at blob rather than links ✓						
#72 by masokuk99 was merged last month • Approved						
Fixed notifications and authentication ✓						
#6 by Denzel09 was merged 27 days ago • Approved						
Middleware and accept tokens when SignalR connects ✓						
#20 by Denzel09 was merged last month • Approved						
modified chatcontroller for users ✓						
#2 by Denzel09 was merged on Oct 29 • Approved						
broadcast telemetry and flag stale vehicles offline ✓						
#23 by masokuk99 was merged on Oct 28 • Approved						
fixed endpoint ✓						
#8 by Denzel09 was merged on Oct 24 • Approved						
hosting database in neon ✓						
#27 by Denzel09 was merged on Oct 24 • Approved						
supporting profile updates ✓						
#20 by masokuk99 was merged on Oct 24 • Approved						
0 Open	✓ 12 Closed					
Added pipeline to frontend ✓						
#12 by Denzel09 was merged 2 days ago						
improved ui across all pages, enabled 2-way connectivity and more sma... ✓						
#11 by masokuk99 was merged 2 days ago • Approved						
optimize chat loading, update navigation, and clean login UI ✓						
#10 by masokuk99 was merged last month • Approved						
f ✓						
#9 by masokuk99 was merged last month • Approved						
Fixed code ✓						
#8 by Denzel09 was merged 27 days ago • Approved						
fixed code ✓						
#7 by Denzel09 was merged last month • Approved						
Fix some code ✓						
#6 by Denzel09 was merged last month • Approved						
Fixed realtime communication between two persons ✓						
#5 by Denzel09 was closed last month • Approved						
added chat functionality and modal ✓						
#4 by Denzel09 was merged on Oct 29 • Approved						
add live fleet map and shared header navigation ✓						
#3 by masokuk99 was merged on Oct 28 • Approved						
revamped navbar, fixed bugs and improved performance ✓						
#2 by masokuk99 was merged on Oct 24 • Approved						
Feature/auth updates and signin-administrator page done ✓						
#1 by masokuk99 was merged on Oct 23 • Approved						

(1) Web API PR-historik

(2) Webapp PR-historik

Figur 6.2: PR historikken for både webappen (1) og Web API'et (2) viser, at ændringer løbende er blevet integreret via PR'S, som efter godkendelse er blevet flettet ind i main-branchen.

7 Analyse

7.1 Nuværende muligheder

I forbindelse med projektets opstart blev der foretaget en analyse af, hvordan flådeovervågning typisk håndteres i dag. Mange virksomheder anvender en kombination af GPS-enheder, simple track-and-trace-systemer eller ikke-integrerede arbejdsgange, hvilket ofte resulterer i et fragmenteret overblik over køretøjerne [4]. For en mere omfattende analyse af nuværende muligheder henvises til bilag 4.a.

Flådekoordinatører mangler generelt et samlet, realtidsopdateret dashboard, der gør det muligt både at visualisere positioner, overvåge hændelser og kommunikere direkte med chauffører. Reeltidsdata har dokumenteret effekt på beslutningstagning, sikkerhed og effektivitet i flåder [1], [2]. Chauffører mangler omvendt en løsning, hvor beskeder, statusopdateringer og ruteændringer sker i ét samlet system i stedet for i flere adskilte applikationer [3].

Selvom et flådestyringssystem i principippet kan understøtte mange forskellige køretøjstyper, håndteres de mere tidskritiske leverancer ofte af lastbiler. Lastbiltransport beskrives som den dominerende godstransportform, netop fordi fleksibel, hurtig *door-to-door*-service gør den særligt velegnet til tidsfølsomme leveringer [5].

I forlængelse af dette designes systemet til at kunne understøtte de fleste køretøjstyper, såsom lastbiler og skibe, men projektets primære fokus er lastbiler, da det er her realtidsopdateringer og hurtig koordinering giver størst praktisk værdi.

7.2 Integration og begrænsninger

En af de største begrænsninger i projektet er manglen på adgang til live-data fra rigtige køretøjer. Derfor anvendes en simulator som datakilde. Dette er en almindelig løsning i projekter, hvor virkelige trackingdata ikke er tilgængelige [4].

Analysen ser også på, hvordan systemet kan designes, så det senere kan integreres med eksisterende flådesystemer, GPS-trackere eller andre datakilder. Det kræver, at Web API'et, datamodellen og de valgte kommunikationsmetoder er fleksible og kan udvides uden større ændringer.

Begrænsningsanalysen er vigtig, fordi den tydeliggør, hvad projektet faktisk kan evaluere. Fokus er ikke på at måle præcisionen af virkelige GPS-data, men på at undersøge, om systemets arkitektur kan leve et samlet realtidsbillede — også når data stammer fra en simulator.

8 Softwarearkitektur

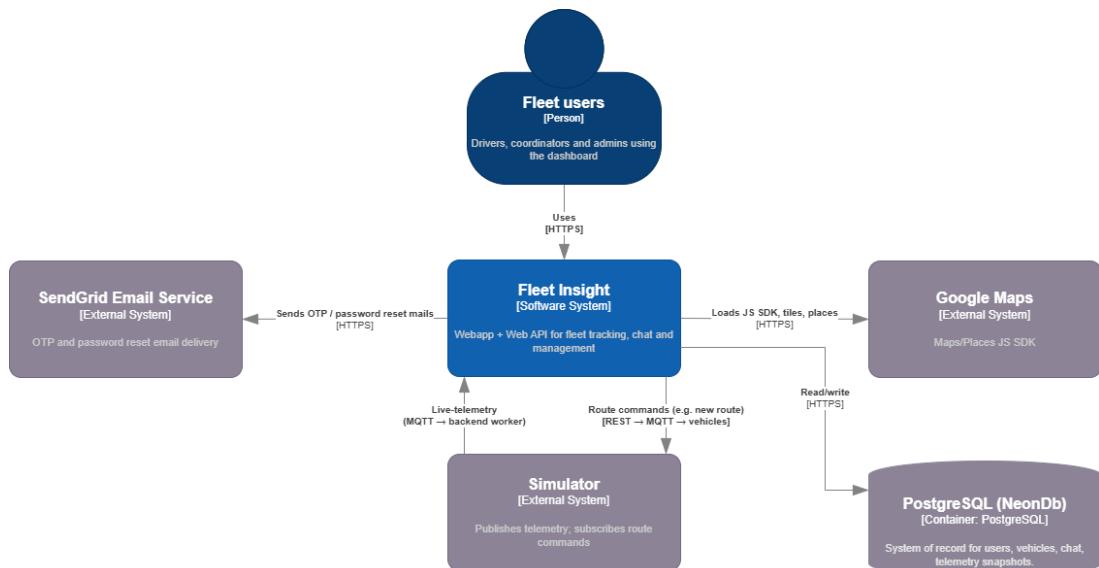
8.1 C4 og systemanalyse

I henhold til metodeafsnittet [6.1.1 Softaredokumentation](#) er softwarearkitekturen modelleret efter C4-modellen. Det indledende skridt var at udforme et kontekstdiagram, som er vist på figur [8.1](#).

8.2 C4 model afgrænsning

I projektet er der bevidst ikke udarbejdet klassediagrammer som del af C4-dokumentationen, da de ikke ville tilføje væsentlig indsigt ud over C3-komponentdiagrammet, som allerede beskriver systemets centrale strukturer og ansvar. Klassediagrammerne ville primært visualisere simple DTO-klasser, som er lette at forstå direkte i koden, og de ville hurtigt blive forældede ved implementeringsændringer og dermed risikere at give misvisende dokumentation.

8.2.1 C1 diagram



Figur 8.1: Kontekstdiagram for Fleet Insight. Systemet er markeret blåt og eksterne aktører gråt. Interaktionerne med eksterne aktører er vist med pile, hvor pilenes retning angiver, hvem der initierer kommunikationen. Fleet users benytter systemet via webappen, mens eksterne systemer som SendGrid, Google Maps, PostgreSQL og simulatoren leverer data og tjenester til systemet.

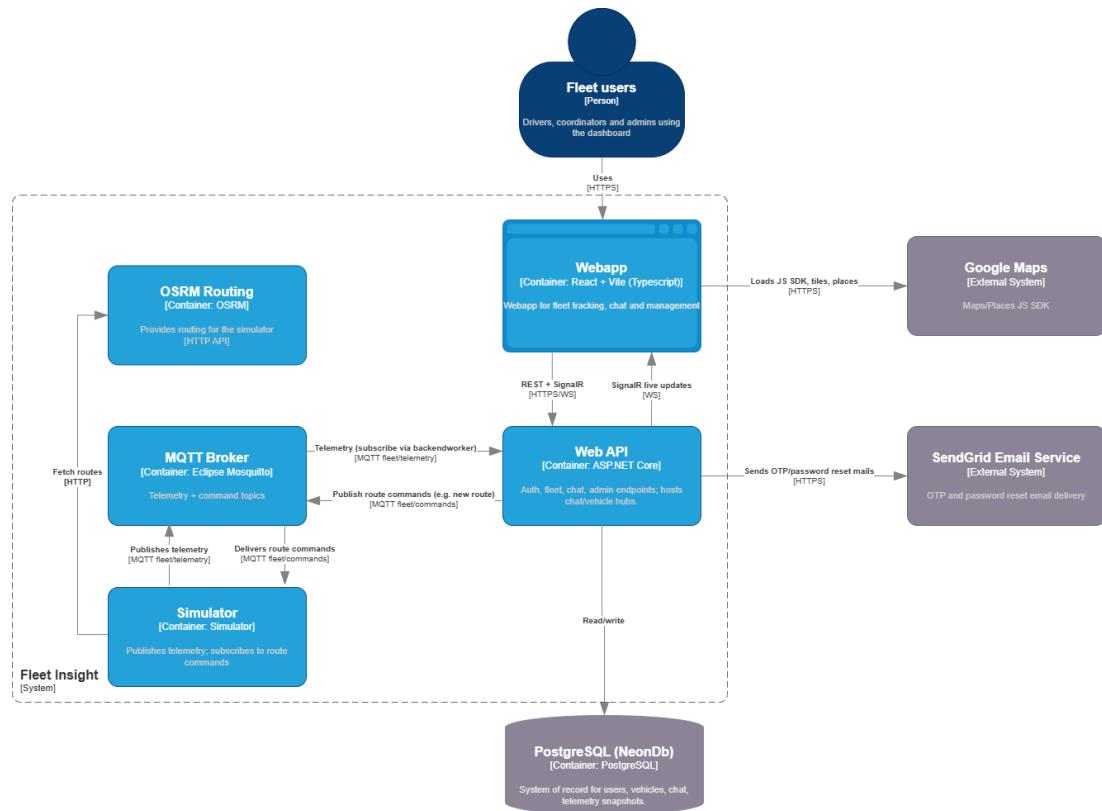
På kontekstdiagrammet ses de aktører og eksterne systemer, som Fleet Insight interagerer med. Fleet users, initierer kommunikationen med systemet gennem webappen, hvilket stiller

krav til arkitekturen og de funktioner, systemet skal understøtte. Flere af disse krav er udledt af projektets kravspecifikation i afsnit [5 Kravspecifikation](#), hvor både funktionelle og ikke-funktionelle behov er beskrevet.

Derudover skal systemet kommunikere med en række eksterne tjenester, herunder Google Maps til kortvisning, SendGrid til udsendelse af engangskoder for sign-up og nulstilling af adgangskode samt en simulator, der understøtter to-vejs kommunikation, der leverer løbende køretøjsdata. Disse integrationer stiller krav til sikkerhed, ydeevne og stabilitet, hvilket også afspejles i de ikke-funktionelle krav i afsnit [5.1 Ikke-funktionelle krav](#).

Sammenfattende giver kontekstdiagrammet et overblik over systemets omgivelser. Det danner dermed grundlaget for det efterfølgende containerdiagram, hvor systemets interne struktur og tekniske opdeling præsenteres på figur [8.2](#).

8.2.2 C2 diagram



Figur 8.2: Containerdiagrammet for *Fleet Insight* viser systemets interne containere og deres interaktioner med eksterne systemer. Den stipede ramme markerer grænsen for *Fleet Insight*, mens alt udenfor rammen repræsenterer eksterne tjenester. Indenfor ses de centrale dele af systemet, herunder webappen, Web API'et og simulatorens infrastruktur, som tilsammen udgør systemets arkitektur.

På containerdiagrammet ses den overordnede arkitektur for Fleet Insight. Den stiplede ramme markerer systemgrænsen, og alle elementer udenfor denne repræsenterer eksterne systemer, som løsningen integrerer med.

Brugeren interagerer udelukkende med webappen, som kommunikerer med Web API'et via HTTPS og WebSockets (SignalR). Webappen har ingen direkte adgang til databasen, MQTT-brokeren, OSRM eller e-mailtjenesten; eneste undtagelse er Google Maps, hvor kortdata og Places-SDK'et hentes direkte i webappen.

Web API'et udgør systemets kerne og håndterer autentifikation, bruger- og flådestyring, chat samt realtidsopdateringer via SignalR. Web API'et står for al kommunikation med systemets eksterne afhængigheder:

- **PostgreSQL (NeonDb)** - systemets primære datalager for brugere, køretøjer, chat og telemetri.
- **SendGrid** - afsendelse af OTP- og password-reset e-mails via HTTPS.

OSRM Routing er en intern container, der udelukkende bruges af simulatoren til at hente ruter til simulatoren; Web API'et kalder den ikke direkte.

Diagrammet viser også en simulator, der fungerer som en intern telemetrikilde og publicerer det via MQTT samt modtager rute-kommandoer fra Web API.

Arkitekturen understøtter de funktionelle krav beskrevet i afsnit [5 Kravspecifikation](#), herunder realtidsopdateret flådevisning, rutehåndtering, sikker login-flow og chat. Containeropdelingen følger principperne fra C4-modellen og sikrer en klar adskillelse mellem webappen, Web API'et og eksterne systemer samt en skalerbar struktur.

8.3 Systemsekvensdiagram

Der er blevet lavet syv systemsekvensdiagrammer, hvilket vil sige en for hver user story. I dette afsnit vil systemsekvensdiagrammerne for *Opret bruger*, *Flåde* og *Chat* vises og beskrives. For at se de andre systemsekvensdiagrammer, se bilag [2.a.a](#). Yderligere er der også blevet lavet syv applikationssekvensdiagrammer, hvilket vil sige en for hver user story, se bilag [2.a.b](#).

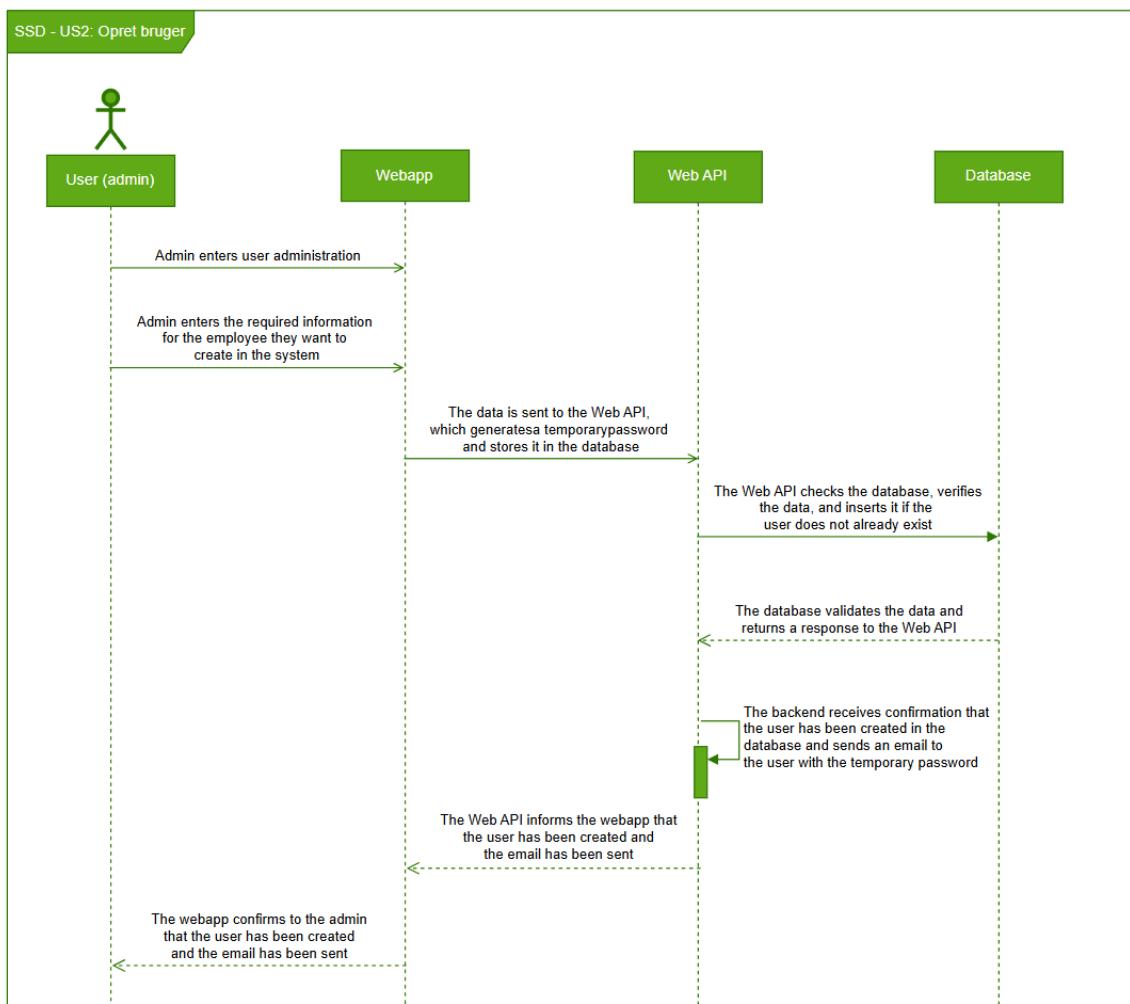
8.3.1 Opret bruger

Når en admin åbner brugeradministrationen, indtastes de nødvendige oplysninger for den medarbejder, der skal oprettes i systemet. Når informationerne er angivet, sender webappen dataene til Web API'et, som genererer en OTP og opretter brugeren i databasen.

Web API'et kontrollerer først i databasen, om e-mailadressen allerede findes. Hvis brugeren ikke eksisterer, indsætter databasen den nye bruger og bekræfter oprettelsen tilbage til Web API'et. Herefter udsender Web API'et en e-mail til medarbejderen med OTP'en, så brugeren kan logge ind.

Når både databaseoprettelse og e-mailafsendelse er gennemført, svarer Web API'et til webappen, at brugeren er oprettet, og at e-mailen er sendt. Webappen giver derefter admin en visuel bekræftelse på, at processen er fuldført.

Systemsekvensdiagrammet viser dermed det primære succesforløb for oprettelse af en ny bruger, fra indtastning i webappen til oprettelsen i databasen og udsendelse af loginoplysninger. Alternative forløb og fejlscenarier — såsom eksisterende bruger eller mailfejl — er ikke vist i diagrammet.

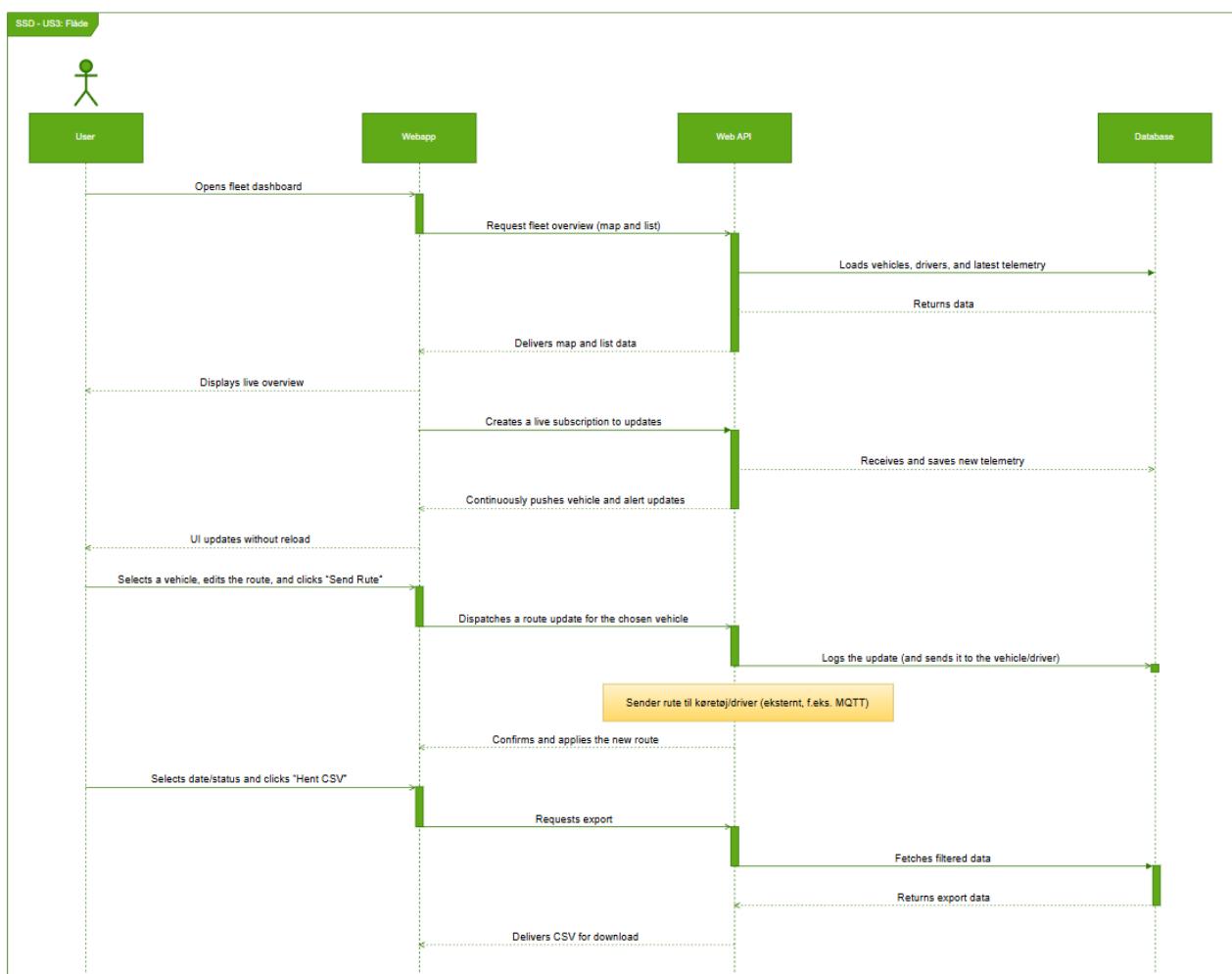


Figur 8.3: Systemsekvensdiagram for oprettelse af brugere, som viser flowet fra validering i webappen til oprettelse i Web API'et og udsendelse af OTP til brugeren

8.3.2 Flåde

På sekvensdiagrammet i figur 8.4 ses hovedforløbet for user story 3 (Flåde). Når brugeren åbner flådesiden, sender webappen en forespørgsel til Web API'et om de nødvendige data til kort og køretøjsliste (telemetridata), som hentes fra databasen og returneres til brugeren. Herefter lytter webappen efter live opdateringer, så ændringer i telemetridata og alarmer kan vises uden at siden genindlæses. Når brugeren ændrer en rute, sendes opdateringen til Web API'et, der gemmer ændringen og sørger for, at den registreres i systemet, hvorefter den nye rute afspejles i brugerens visning. Ved eksport af data sender webappen en forespørgsel til Web API'et, som henter og filtrerer de relevante data i databasen og returnerer dem som en CSV-fil til download.

Systemsekvensdiagrammet illustrerer altså det overordnede flow mellem bruger, webapp, Web API og database i forbindelse med visning af flåden. Diagrammet viser, hvordan data hentes, opdateres i realtid og behandles ved ruteændringer og dataeksport.



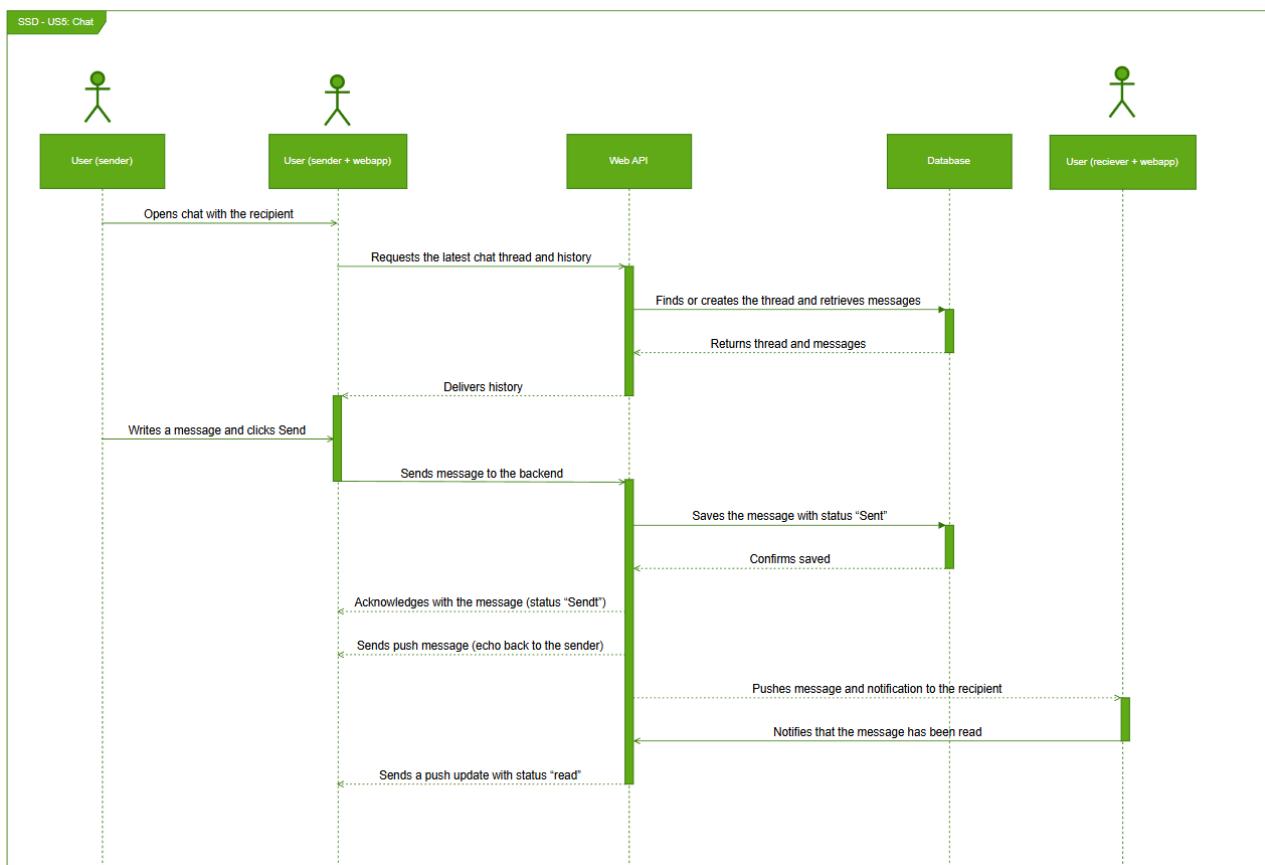
Figur 8.4: Systemsekvensdiagram for flåde, der viser forløbet for datahentning, live-opdateringer, ruteændringer, alarmer og CSV-eksport

8.3.3 Chat

På figur 8.5 ses sekvensforløbet for chatfunktionen. Inden sekvensen starter, forudsættes det, at brugeren er logget ind og befinner sig på chatsiden.

Når afsenderen åbner en chat, anmoder webappen om den tilhørende tråd og tidligere beskeder. Web API'et finder eller opretter den relevante tråd og returnerer historikken til klienten. Når afsenderen sender en ny besked, gemmes den i databasen, hvorefter Web API'et distribuerer beskeden til både afsender og modtager. Modtageren får vist beskeden samt en notifikation. Når en besked bliver læst, sendes dette tilbage til Web API'et, som derefter opdaterer afsenderens visning.

Systemsekvensdiagrammet illustrerer altså et almindeligt succesforløb for 1:1-chatten. Fejlhåndtering er ikke vist, da fokus er på det primære flow.



Figur 8.5: Systemsekvensdiagram for 1:1-chatfunktionen, som viser forløbet fra hentning af tråd og historik til afsendelse, modtagelse og læsning af beskeder

8.4 CI pipeline og Build-proces

Som en del af udviklingsprocessen er der implementeret en CI pipeline, som automatisk bygger Web API'et og tester alle repositories. Formålet er at sikre, at ændringer valideres løbende, så fejl opdages tidligt og kodekvaliteten fastholdes.

CI-pipelinens sørger for en stabil udviklingsproces ved automatisk at bygge projektet og køre relevante tests, hver gang der foretages ændringer i koden. På den måde fungerer pipelinens som en kontinuerlig kvalitetskontrol, der sikrer, at kun testet og valideret kode integreres i projektet.

Pipelinen kompilerer projektet og afvikler automatiserede tests, hvilket er fundamentet for kvalitetssikringen af Web API'et. Testprocessen beskrives mere detaljeret i afsnit [11.2 Test af Web API](#).

Ingen miljøer

Projektet inkluderer ikke et deployment-step, da der ikke findes et test- eller produktionsmiljø at deploye til. Derfor er der ikke etableret en egentlig deploymentstrategi, og CI-pipelinens fokuserer udelukkende på at bygge og teste koden.

Projektet befinner sig i et lokalt MVP-stadie, hvor en fuld miljøkæde (dev/test/staging/prod) ikke ville give reel værdi. Opsætning af sådanne miljøer ville kræve betydelige ressourcer til hosting, drift og håndtering af secrets, hvilket ligger uden for projektets rammer.

Derfor er der valgt en enklere løsning: ingen deploymentmiljøer, men en solid CI-proces, der automatisk bygger og tester koden og dermed leverer den nødvendige kvalitetssikring uden ekstra kompleksitet.

8.5 Valg af teknologi

I dette afsnit er der blevet foretaget analyse af de forskellige delsystemer, pånær databasen. For at se analysen af databasen, se bilag [4.b](#).

8.5.1 Simulator teknologianalyse

Formålet med simulatoren er at generere realistiske ruter og telemetri, så flådestyring, hændelser og ruteopdateringer kan testes uden adgang til fysiske køretøjer. I stedet for at sammenligne hele teknologistakken under ét, sammenlignes de centrale teknologikomponenter enkeltvist: *programmeringssprog, kommunikationsprotokol og routingmotor*.

Teknologi	Fordele	Ulemper
Sprogvalg		
Python	<ul style="list-style-type: none"> + Hurtig udvikling + Stort økosystem + Let at tilpasse scenarier 	<ul style="list-style-type: none"> - Lav performance ift. kompilerede sprog - Mindre egnet til højfrekvente simulationer
Go / C++	<ul style="list-style-type: none"> + Høj performance + Velegnet til store data-mængder 	<ul style="list-style-type: none"> - Længere udviklingstid - Mindre fleksibel
Kommunikationsprotokol		
MQTT	<ul style="list-style-type: none"> + Letvægts pub/sub + Skalerbar ved mange enheder + Standard i IoT telemetri 	<ul style="list-style-type: none"> - Kræver broker - Ikke udbredt i browsers direkte
WebSockets / polling	<ul style="list-style-type: none"> + Simpelt at integrere + Direkte kommunikation 	<ul style="list-style-type: none"> - Dårligere skalerbarhed end MQTT - Polling øger belastning
Routing / Maps		
OSRM	<ul style="list-style-type: none"> + Lokal kontrol + Ingen API omkostninger + Hurtig routing 	<ul style="list-style-type: none"> - Kræver drift af routing-motor - Manuelle kortopdateringer
Hosted routing API	<ul style="list-style-type: none"> + Ingen drift + Altid opdaterede kort 	<ul style="list-style-type: none"> - API kvoter og omkostninger - Afhængighed af tredje-part

Tabel 8.1: Opdelt sammenligning af teknologivalg til simulatoren

På baggrund af analysen er Python, MQTT og OSRM valgt for at maksimere udviklingshastighed, fleksibilitet og lokal kontrol over routingdata. Python giver et hurtigt iterativt udviklingsforløb [6], OSRM muliggør lokal og omkostningsfri routing med høj performance [7], og MQTT giver en effektiv og skalerbar push-model til telemetri [8], [9].

8.5.2 Web API teknologianalyse

Der findes mange teknologier, der kan anvendes til udvikling af Web API'er. I dette projekt er fokus blevet indskrænket til to kandidater: ASP.NET Core og Node.js. For at se en dybdegående analyse af dette, se bilag 4.b

På tabel 8.2 ses en sammenfatning af de primære forskelle, fordele og ulemper mellem ASP.NET Core og Node.js [10], [11].

Mulighed	Fordele	Ulemper
ASP.NET	+ Erfaring i teamet + Kortere udviklingstid ved komplekse systemer + Typesikkert og stærkt struktureret sprog	- Kræver mere ressource-tung runtime end lettere alternativer - Mindre dynamisk og fleksibelt end JavaScript-baserede løsninger - Tæt knyttet til Microsofts økosystem
Node.js	+ Hurtig eksekvering pga. V8-motoren + Dynamisk og fleksibelt sprog	- Større risiko for runtime-fejl pga. manglende typesikkerhed - Kan være sværere at vedligeholde i større projekter - Stor afhængighed til tredjepartsbiblioteker med varierende kvalitet

Tabel 8.2: Sammenligning af ASP.NET og Node.js

På baggrund af analysen har projektgruppen valgt at anvende ASP.NET Core som fundament for Web API'et. Valget skyldes blandt andet .NET-platformens typestærke og eksplisitte syntaks, som bidrager til en mere stabil og struktureret udviklingsproces. Samtidig har teamet allerede erfaring med ASP.NET Core, hvilket yderligere understøtter beslutningen.

8.5.3 Webapp teknologianalyse

I udviklingen af webappen af systemet blev der foretaget en vurdering af relevante teknologier. For at se en mere gennemgående analyse af dette, se bilag 4.b.

På tabel 8.3 ses en oversigt over de centrale forskelle, styrker og svagheder ved React og Angular, hvor de vigtigste fordele og ulemper er sammenfattet [12], [13].

Mulighed	For (Fordele)	Imod (Ulemper)
React (TypeScript)	+ Stor fleksibilitet og letvægtsbibliotek + Stort økosystem og bred community-support + TypeScript giver stærkere typesikkerhed og færre runtime-fejl	- Kræver mange eksterne pakker for fuld funktionalitet - Mangler officielle standarder for struktur - Stejl læringskurve for begyndere pga. mange valg
Angular (TypeScript)	+ Kompleks framework-løsning med indbygget router, DI, formularhåndtering m.m. + Stærk struktur og opinionated arkitektur + TypeScript som standard	- Tungere og større bundle-size end React - Stejl indlæringskurve pga. kompleksitet - Mindre fleksibelt, da mange valg er låst af frameworket

Tabel 8.3: Sammenligning af React (TypeScript) og Angular

På baggrund af analysen er React med TypeScript valgt som webapp-teknologi til dette projekt. Valget skyldes især kombinationen af fleksibilitet, stærk community-støtte samt forbedret typesikkerhed gennem TypeScript, som samlet bidrager til en mere effektiv og stabil udviklingsproces.

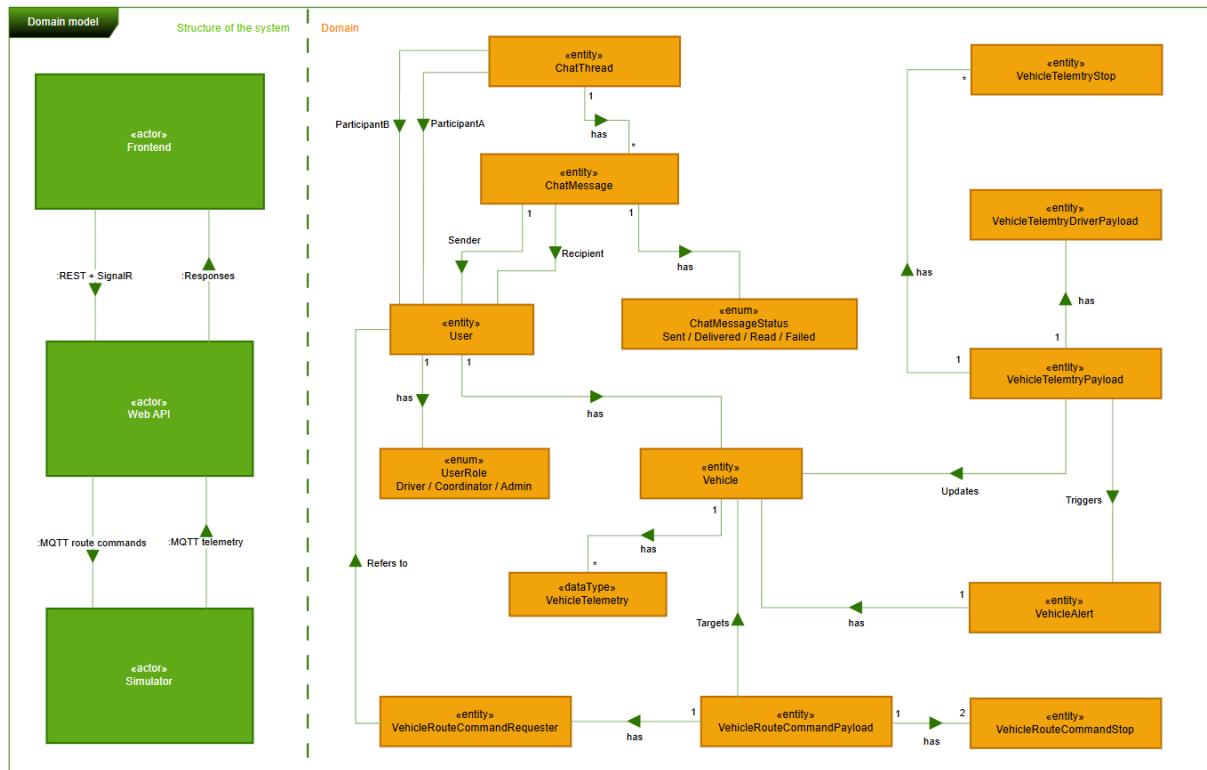
9 Software Design

9.1 Domæne model

Nedenfor på figur 9.1 er domænemodellen for systemet givet, og den illustrerer, hvordan de centrale entiteter hænger sammen. Domænemodellen giver et samlet overblik over de centrale begreber i systemet og de indbyrdes relationer mellem dem. Modellen viser, hvordan chatfunktionalitet, brugerroller, køretøjer, telemetri, ruteopdateringer og alarmer hænger sammen i et fælles domæne.

Den illustrerer samtidig, hvordan data bevæger sig gennem systemet — fra brugernes interaktioner, over beskeder og køretøjer, til telemetri, ruteændringer og de hændelser, der kan opstå som følge heraf.

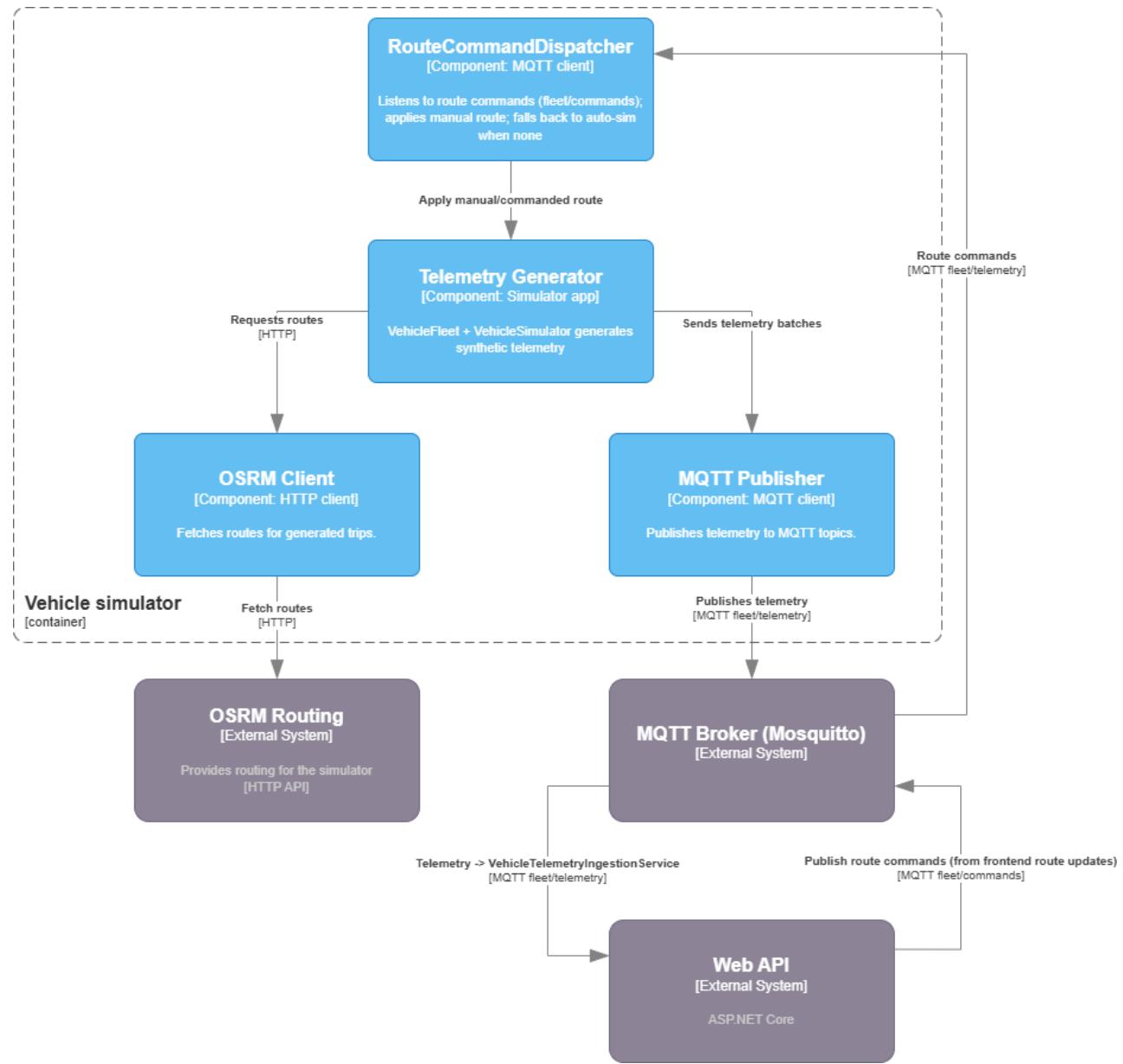
Formålet er at skabe en fælles forståelse af systemets kerneobjekter og deres forbindelser på et overordnet niveau, så arkitektur, implementering og krav kan tage udgangspunkt i et konsistent domæne.



Figur 9.1: Samlet overblik over systemet. Venstre side viser den overordnede kommunikation mellem delsystemerne, mens højre side præsenterer domænemodellen med systemets centrale begreber og deres indbyrdes relationer. Modellen danner grundlaget for systemets struktur, dataflow og funktionelle sammenhæng.

9.2 Simulator design

Figur 9.2 viser C3-komponentdiagrammet for simulatoren. Simulatoren fungerer som et selvstændigt system, der genererer telemetridata for virtuelle køretøjer baseret på ruter fra et vejnet og kommunikerer med Web API'et via MQTT. Systemet understøtter både automatisk rutesimulering og manuelle ruteopdateringer initieret fra webappen.



Figur 9.2: C3-komponentdiagram for simulatoren. Simulatoren er opdelt i selvstændige komponenter til rutehåndtering, simulering og kommunikation. Telemetry Generator simulerer køretøjsbevægelser baseret på ruter fra OSRM, manuelle ruteændringer modtages via RouteCommandDispatcher over MQTT, og genereret telemetri publiceres til Web API'et via MQTT Publisher.

Simulatorens interne komponenter

Telemetry Generator udgør simulatorens kerne og er ansvarlig for selve simuleringen af køretøjerne. Komponenten vedligeholder den interne tilstand for hvert køretøj og genererer løbende telemetridata såsom position, hastighed og bevægelse langs en rute. Når der skal startes nye ture, anmoder Telemetry Generator OSRM Client om rutedata og anvender disse til at beregne realistiske kørselsforløb over tid.

RouteCommandDispatcher fungerer som bindeledd mellem Web API'et og simulatorens kørselslogik. Komponenten lytter på MQTT-topicet `fleet/commands` via MQTT-brokeren. Når en koordinator foretager en manuel ruteændring i webappen, publicerer Web API'et en rute-kommando, som RouteCommandDispatcher modtager og anvender på det relevante køretøj. Hvis der ikke modtages manuelle kommandoer, foretages der automatisk fallback til simulatorens indbyggede autosimulation.

OSRM Client fungerer som en dedikeret HTTP-klient til den eksterne OSRM Routing-service. Komponenten anvendes af Telemetry Generator til at hente detaljerede ruter baseret på vejnettet og sikrer dermed, at simulatorens kørselsmønstre følger realistiske vejforløb fremfor tilfældige eller forenklede bevægelser.

MQTT Publisher er en selvstændig komponent, der er ansvarlig for publicering af telemetridata til MQTT-brokeren. Telemetrimeddelelser sendes til topicet `fleet/telemetry`, som Web API'ets VehicleTelemetryIngestionService abonnerer på. Publiceringen foregår asynkront via en intern kø og en dedikeret worker, hvilket sikrer stabil levering af telemetri uden at blokere simulatorens hovedlogik.

Eksterne systemer

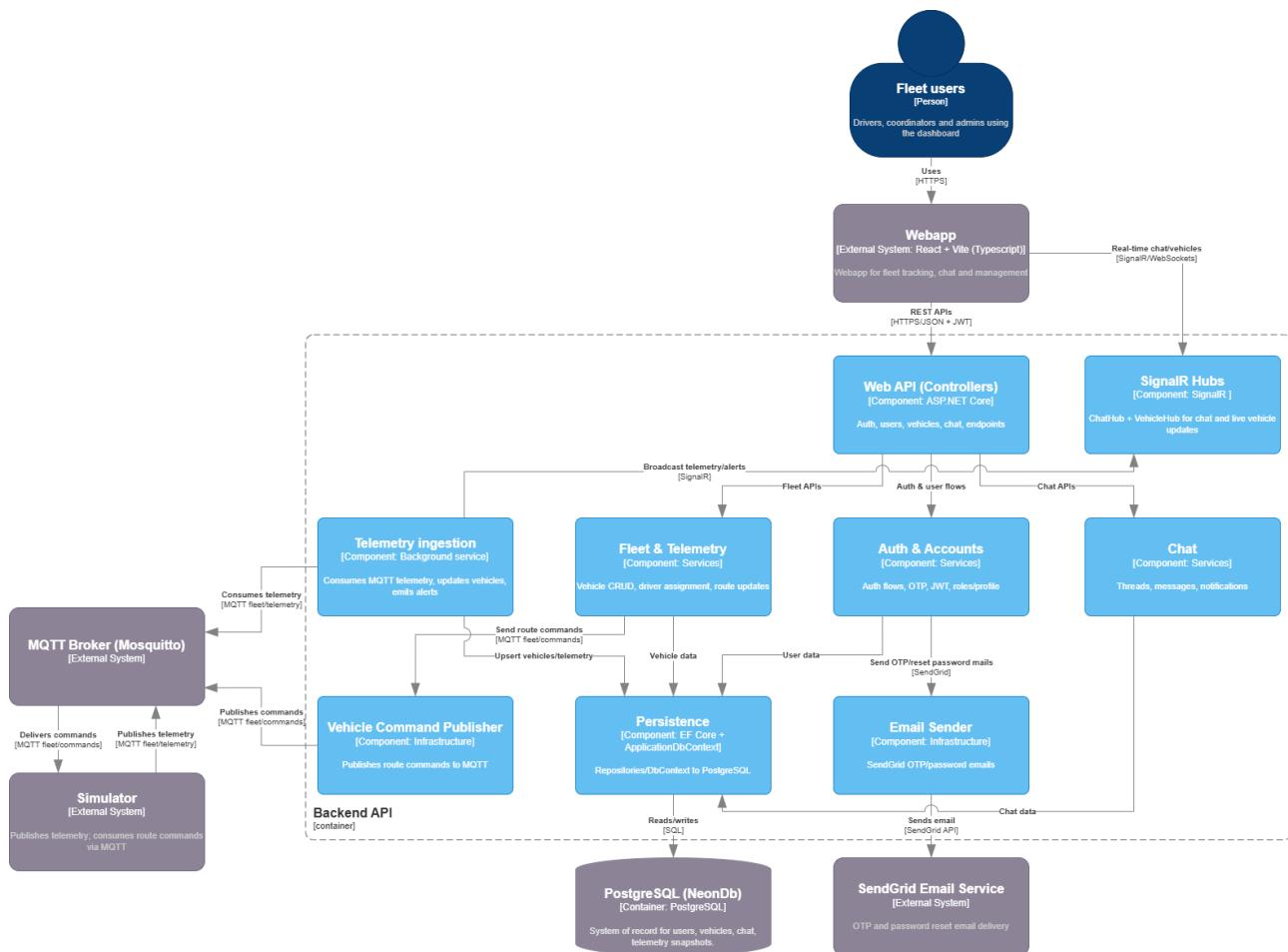
Simulatoren interagerer med følgende eksterne systemer:

- **OSRM Routing:** Ekstern routingservice, der leverer rutedata via et HTTP API.
- **MQTT Broker (Mosquitto):** Formidler al MQTT-baseret kommunikation, herunder rute-kommandoer fra Web API'et og telemetri fra simulatoren.
- **Web API / VehicleTelemetryIngestionService:** Abonnerer på `fleet/telemetry`-topicet og indsætter modtaget telemetridata i systemets backend med henblik på videre behandling og visning i webappen.

9.3 Web API design

Figur 9.3 viser C3-komponentdiagrammet for Web API’et. Diagrammet er forenklet for at give et tydeligt overblik over systemets vigtigste dele og deres indbyrdes afhængigheder. En fuld visning af alle interne komponenter ville gøre diagrammet unødig komplekst.

Web API’et er bygget op som en lagdelt arkitektur: præsentationslaget håndterer HTTP- og realtime-kommunikation, logiklaget indeholder forretningslogikken, og infrastruktur- og datalaget står for databaseadgang og integrationer til eksterne systemer. Afhængighederne går kun én vej — oppefra og ned — hvilket giver en klar ansvarsfordeling og gør systemet lettere at vedligeholde.



Figur 9.3: C3-komponentdiagram for Web API’et. Klientkald håndteres i controllers og SignalR-hubs, behandles i servicekomponenter og persisteres via Persistence-komponenten eller videresendes til eksterne systemer.

Præsentationslaget

Præsentationslaget består af Web API'ets controllers samt SignalR-hubs, som tilsammen udgør systemets indgangspunkter. Controllers udstiller REST-endpoints til autentifikation, flådestyring, køretøjer og chat. Alle kald valideres ved hjælp af JWT og rollebaserede claims, hvorefter de videresendes til logiklaget.

SignalR-hubsene, herunder *ChatHub* og *VehicleHub*, anvendes til realtidskommunikation med webappen. De er ansvarlige for at udsende chatbeskeder, telemetri og alarmer som reaktion på hændelser i systemet, men indeholder hverken forretningslogik eller direkte dataadgang. For at se en mere detaljeret sekvens af hvordan SignalR anvendes via observer-pattern, se bilag [3.a](#).

Logiklaget

Logiklaget indeholder systemets centrale domænelogik og er organiseret i forskellige servicekomponenter. Services under *Auth* og *Accounts* håndterer loginflows, OTP-verificering, bruger- og rolleadministration samt udstedelse af JWTs. Services under *Fleet* og *Telemetry* står for logik relateret til køretøjer, driver-assignments, ruteopdateringer og behandling af telemetridata. *Chat*-servicen håndterer tråde, beskeder, notifikationer og læsekvitteringer.

Logiklaget benytter infrastrukturkomponenterne til persistens, messaging og eksterne integrationer, men indeholder ikke selv disse funktioner.

Infrastruktur- og datalag

Infrastruktur- og datalaget håndterer systemets data og kommunikationen med eksterne tjenester. Komponenten *Persistence* (EF Core + ApplicationDbContext) fungerer som dataadgangslag og står for læse- og skriveoperationer til PostgreSQL-databasen, som lagrer brugere, køretøjer, chatbeskeder og telemetridata. Repositories og ApplicationDbContext er en del af Persistence-komponenten og anvendes af servicelaget for at sikre en struktureret og ensartet dataadgang.

Derudover indeholder laget en baggrundsservice til telemetrihåndtering, *Telemetry Ingestion*, som lytter på MQTT-topicet **fleet/telemetry**, opdaterer databasen og udløser alarmer, der videresendes via SignalR. Til udgående kommandoer benyttes *Vehicle Command Publisher*, som publicerer rute- og køretøjskommandoer på MQTT-topicet **fleet/commands**. Endelig anvendes *Email Sender*-komponenten til integration med SendGrid i forbindelse med udsendelse af OTP- og password-reset-mails.

Eksterne systemer

Som vist i komponentdiagrammet interagerer Web API'et med følgende eksterne systemer:

- **Webapp** - React/Vite-klient, som anvender REST og SignalR til kommunikation med Web API'et.
- **Simulator** - ekstern komponent, der publicerer telemetri og konsumerer køretøjskommandoer via MQTT.
- **PostgreSQL (NeonDB)** - primært datalager.
- **MQTT Broker (Mosquitto)** - formidler telemetri og køretøjskommandoer mellem systemets komponenter.
- **SendGrid Email Service** - anvendes til e-maillevering.

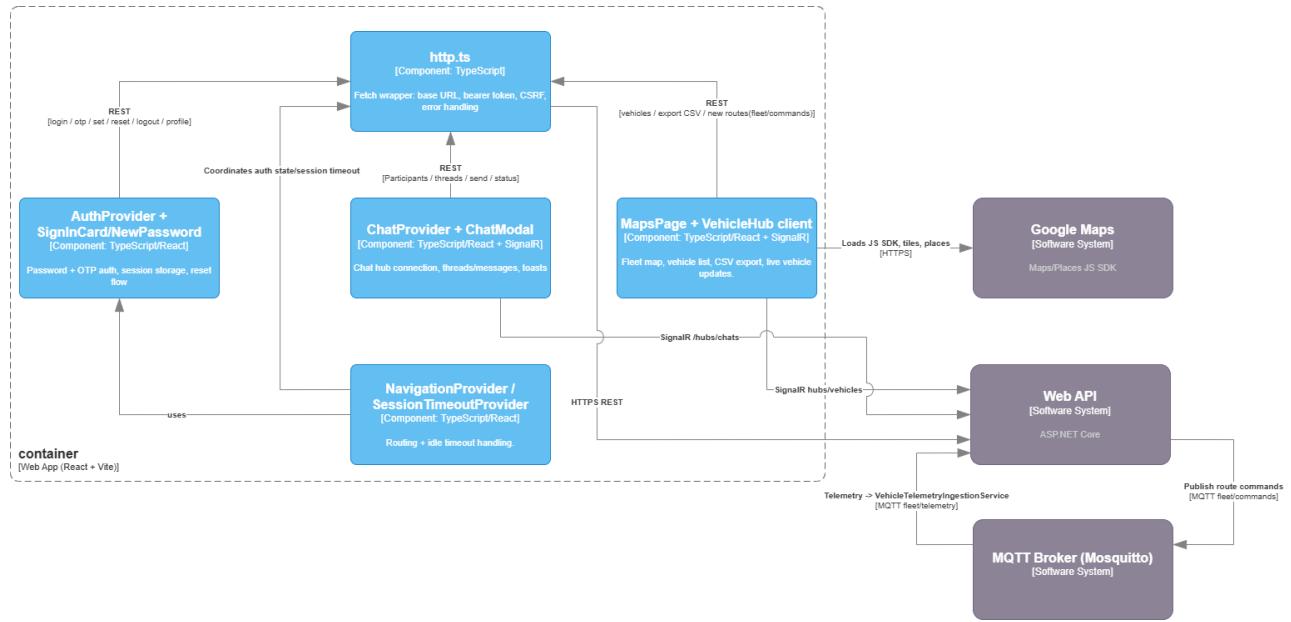
Overordnet dataflow

Et typisk request-flow følger sekvensen: webapp → controller → service → repository, hvorefter et svar returneres til klienten.

9.4 Webapp design

Webappen fungerer som brugergrænseflade for flådestyring, chat og realtime-overvågning. Figur 9.4 viser et bevidst forenklet C3-komponentdiagram for webappen, udformet for at give et klart overblik over de centrale komponenter og deres samspil med Web API'et, SignalR-hubs og eksterne services.

Selvom diagrammet ikke eksplisit opdeler arkitekturen i lag, beskrives webappen i dette afsnit ved hjælp af et *containerlag*, der håndterer fælles teknisk infrastruktur, og et *komponentlag*, som indeholder de UI-moduler, brugeren interagerer med. Denne opdeling sikrer en konsistent og modulær arkitektur.



Figur 9.4: C3-komponentdiagram for webappen. Containerlaget håndterer routing, session, HTTP-kald og SignalR-forbindelser, mens komponentlaget indeholder brugerrettede UI-moduler.

Komponentlaget

Komponentlaget indeholder webappens UI-komponenter og providers, der tilsammen implementerer systemets funktionalitet.

AuthProvider med tilhørende login-, OTP- og reset-komponenter håndterer autentifikationsflow, sessionstilstand og nulstilling af adgangskoder. *ChatProvider* og *ChatModal* implementerer chatfunktionaliteten og anvender REST-kald til hentning af deltagere, tråde og historik samt realtime-opdateringer via SignalR-hubben `/hubs/chat`.

Flådefunktionerne håndteres af *MapsPage* i kombination med en *VehicleHub*-klient. *MapsPage* anvendes til visning af flåden på kort, livepositionsopdateringer, CSV-eksport af køretøjsdata samt håndtering af ruteændringer. Statiske data og brugerinitierede handlinger tilgår Web API’et via REST, mens realtime-opdateringer modtages via SignalR-hubben `/hubs/vehicles`. Ruteændringer sendes til Web API’et, som videreforsynder dem til simulatoren via MQTT; webappen kommunikerer aldrig direkte med MQTT.

Google Maps indlæses via Maps/Places JavaScript SDK og anvendes af *MapsPage* til kortvisning

Containerlaget

Containerlaget udgør webappens fælles tekniske fundament og indeholder generel infrastruktur, som genbruges af UI-komponenterne.

Komponenten `http.ts` fungerer som et centralt fetch-wrapper for alle REST-kald og håndterer base-URL, JWT-token, CSRF-header samt ensartet fejlhåndtering.

SignalR-klienterne til chat og køretøjer er placeret i containerlaget og håndterer forbindelser til Web API'ets hubs (`/hubs/chat` og `/hubs/vehicles`) samt reconnect-logik. Navigation og session-timeout håndteres af *NavigationProvider* og *SessionTimeoutProvider*.

Ved at samle netværk, realtime-kommunikation og sessionstyring i containerlaget undgås gentagelse af teknisk infrastruktur i UI-komponenterne.

Eksterne systemer

Webappen integrerer med følgende eksterne systemer:

- **Web API og SignalR** - leverer REST-endpoints og realtime-opdateringer via `/hubs/chat` og `/hubs/vehicles`.
- **Google Maps Platform** - anvendes til kort, tiles og places via Maps/Places JavaScript SDK.
- **MQTT Broker (Mosquitto)** - tilgås indirekte gennem Web API'et til ruteopdateringer og telemetri.

Den klare adskillelse mellem containerlag og komponentlag gør webappen overskuelig, vedligeholdbar og let at udvide, idet teknisk infrastruktur er adskilt fra brugerrettet funktionalitet.

10 Software implementering

I dette afsnit vil implementeringen for de forskellige delsystemer blive beskrevet. Softwaren er udviklet gennem de forskellige sprints, som illustreret på figur 4.2. For en samlet gennemgang af hvordan systemet køres, henvises der til bilag 5.j.

10.1 Simulator implementering

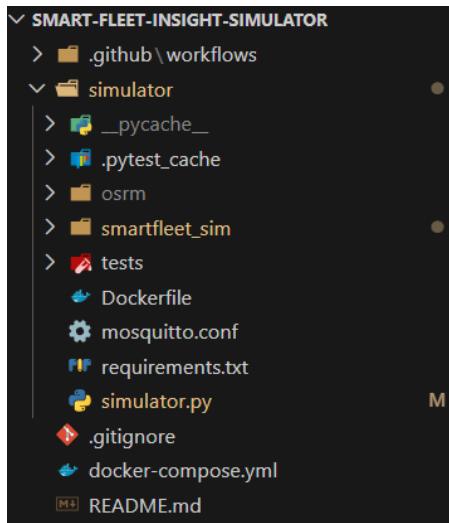
Simulatoren fungerer som et virtuelt flådesystem, der genererer realistisk, kunstig telemetri for køretøjer i realtid. Som udgangspunkt danner simulatoren selv ruter ved hjælp af OSRM, som beregner ruter på et lokalt Danmark-udtræk fra OpenStreetMap leveret af Geofabrik [14]. I praksis afvikles OSRM i en særskilt Docker-container baseret på projektets officielle OSRM-backend-image [15], mens MQTT-brokeren kører i den officielle Eclipse Mosquitto Docker-image [16]. Detaljeret gennemgang af, Docker, MQTT og OSRM-setup, geodætiske beregninger og konfiguration findes i bilag 5.e.

Ud over de automatisk genererede ruter kan simulatoren også modtage manuelle ruteændringer. Disse kommer fra webappen, sendes videre gennem Web API'et og ender til sidst i simulatoren via MQTT, hvor de erstatter køretøjets nuværende rute. Simulatoren understøtter dermed både autonom rutegenerering og direkte styring fra brugeren.

Den fulde sourcekode til simulator findes i bilag 5.c

10.1.1 Simulator folder struktur

Figur 10.1 viser simulatorens overordnede projektstruktur. Logikken ligger samlet i én kerne-mappe `smartfleet_sim`, som står for rutehåndtering, telemetrigenerering og kommunikation via MQTT. OSRM-mappen indeholder de preprocessede vejnetdata for Danmark, som er genereret ved hjælp af OSRM's officielle guide [17]. Udelukkende er Danmark blevet valgt, for at holde ressourcerækket nede, da selv ét land fylder betydeligt i RAM. Docker- og configurationsfiler gør det muligt at køre simulator, OSRM-router og Mosquitto-broker isoleret i hver sin container, men koblet via et fælles Docker-netværk. Strukturen er holdt enkel for at sikre et gennemskueligt projekt, og den fulde opsætning uddybes i bilag 5.e.



Figur 10.1: Oversigt over simulatorens projektstruktur, herunder kernekasse, OSRM-data og konfiguration.

10.1.2 Generering af køreprofil

På Figur 10.2 ses funktionen `build_track`. Den omsætter en rå rute fra OSRM til en detaljeret sekvens af telemetripunkter, som køretøjet bevæger sig igennem.

Først beregnes afstanden mellem alle punkter ved hjælp af en haversine-funktion (afstand på Jordens overflade) og en bearing-funktion, der angiver retningen mellem punkterne. Havserine-implementeringen følger den standardiserede geodætiske formel til afstandsberegning på en kugle, som bl.a. er beskrevet i [18]. Disse beregninger anvendes både til at kontrollere OSRM's rapporterede distance (plausibilitetstjek og skalering) og til at sikre, at ruten ikke fremstår urealistisk, før den omsættes til telemetri. En mere teknisk gennemgang af havserine-, bearing- og distancevalideringen findes i bilag 5.e.

Herefter deles hvert rutesegment op i mange små bevægelsespunkter, hvor hvert punkt repræsenterer køretøjets position ved en given telemetriopdatering. For hvert punkt beregnes bl.a. position, fart, tilbagelagt distance, resterende distance og samlet fremdrift. Funktionen tilføjer også realistiske variationer såsom små fartudsving, korte perioder med overhast samt punkter hvor køretøjet holder stille (idle). Til sidst beregnes et *ETA*-felt, som i implementeringen er baseret på antallet af resterende steps multipliceret med `publish_interval` — ikke ud fra resterende distance og aktuel hastighed.

```

entry = {
    "lat": lat,
    "lon": lon,
    "bearing": bearing,
    "speed_kmh": step_speed_kmh,
    "distance_travelled_m": distance_travelled,
    "distance_remaining_m": distance_remaining,
    "progress": progress,
    "delta_m": step_distance,
}
track.append(entry)
if idle_probability > 0.0 and random.random() < idle_probability:
    track.append(
        {
            "lat": lat,
            "lon": lon,
            "bearing": bearing,
            "speed_kmh": 0.0,
            "distance_travelled_m": distance_travelled,
            "distance_remaining_m": distance_remaining,
            "progress": progress,
            "delta_m": 0.0,
        }
    )
cumulative_distance += segment_distance
if not track:
    raise RuntimeError("Unable to generate track samples for route")
track[-1].update(
{
    "lat": route_coords[-1][0],
    "lon": route_coords[-1][1],
    "speed_kmh": 0.0,
    "distance_travelled_m": total_distance,
    "distance_remaining_m": 0.0,
    "progress": 1.0,
})
total_steps = len(track)
for idx, entry in enumerate(track):
    remaining_steps = total_steps - idx - 1
    entry["eta_seconds"] = remaining_steps * publish_interval
return track

```

Figur 10.2: Kerne-logikken i `build_track`, som danner en detaljeret bevægelsesprofil ud fra OSRM's rutegerometri.

10.1.3 Periodisk udsendelse af telemetri

På Figur 10.3 ses simulatorens telemetriske loop. Denne funktion kører kontinuerligt og udsender én telemetripakke per køretøj ved hvert interval.

Før løkken starter, sikrer simulatoren, at OSRM er tilgængelig ved at kalde `wait_for_osrm`, som først giver grønt lys, når routingmotoren svarer korrekt. I hver iteration beregnes alle køretøjers næste datapunkt parallelt i en threadpool via `self._executor.map`, hvilket producerer ét payload per køretøj. Payloads, der returnerer `None`, bliver automatisk frasorteret, så kun gyldige

datapakker sendes videre.

De genererede telemetripakker publiceres herefter via en intern MQTT-publisher til det konfigurerede telemetri-topic på Mosquitto-brokeren, der kører i en separat container baseret på det officielle Docker-image [16]. Efter hver iteration udføres et *flush*, så alle beskeder er sendt, inden cyklussen afsluttes.

Til sidst beregner simulatoren, hvor meget af intervallet der er tilbage. Jitter og søvn udføres kun, hvis der faktisk er resttid (`base_sleep > 0`). I tilfælde hvor beregningen overstiger intervallet, springes søvn helt over. Hvis simulatoren både har resttid og `publish_jitter > 0`, tilføjes et lille tilfældigt delay, mens standardkonfigurationen uden jitter giver en fuldt deterministisk afsendelsesfrekvens. Detaljer om threadpool, QoS og fejlstrategi er beskrevet i bilag 5.e.

```
def run(self) -> None:
    if not self._vehicles:
        print("No vehicles initialised; exiting.", flush=True)
        return

    if not wait_for_osrm(self.config):
        print("OSRM service did not become ready; exiting.", flush=True)
        return

    print(
        f"Publishing telemetry for {len(self._vehicles)} vehicle(s) to topic '{self.config.mqtt_topic}'"
        f"on {self.config.mqtt_host}:{self.config.mqtt_port}",
        flush=True,
    )

    try:
        while not self._stop.is_set():
            loop_start = time.perf_counter()
            frames: List[Optional[Dict[str, object]]] = list(
                self._executor.map(lambda vehicle: vehicle.next_payload(), self._vehicles)
            )
            for payload in frames:
                if not payload:
                    continue
                self.publisher.publish_payload(payload)
                self._log_payload(payload)
            self.publisher.flush()
            elapsed = time.perf_counter() - loop_start
            base_sleep = max(0.0, self.config.publish_interval - elapsed)
            if base_sleep > 0:
                jitter = 0.0
                if self.config.publish_jitter > 0.0:
                    jitter = random.uniform(-self.config.publish_jitter, self.config.publish_jitter)
                time.sleep(max(0.0, base_sleep + jitter))
        except KeyboardInterrupt:
            print("Simulation interrupted by user.", flush=True)
    finally:
        self._stop.set()
        self._executor.shutdown(wait=True)

def stop(self) -> None:
    self._stop.set()
```

Figur 10.3: Telemetriloopet, som periodisk henter næste datapunkt for alle køretøjer og publicerer dem via MQTT.

10.1.4 Manuelle ruteændringer via MQTT

På Figur 10.4 ses funktionen `handle_message`, som håndterer manuelle ruteændringer sendt fra webappen. Web API’et videresender ændringen til simulatoren via MQTT, og funktionen behandler derfor udelukkende det manuelle flow — den påvirker ikke simulatorens automatisk genererede ruter.

Funktionen starter med at kontrollere, om beskedens topic matcher det konfigurerede kommando-topic. Herefter forsøges beskeden dekodet via `_decode_payload`. Denne funktion returnerer `None`, hvis beskeden ikke kan parses, og sådanne beskeder afvises uden yderligere behandling. Der udføres altså ikke selvstændig valideringslogik i selve handleren. Når payloaden er gyldig, filtreres den efter type (f.eks. `set_route`, `assign_route`, `update_route`), så kun relevante ruteopdateringer fortsætter til næste trin.

Det tilhørende køretøj findes via `_resolve_vehicle`, som søger baseret på id eller nummerplate. Stoplisten parses ved `_parse_stops`, som returnerer `None`, hvis koordinaterne er ugyldige — i så fald droppes beskeden. Værdier som `base_speed_kmh` læses fleksibelt gennem `read_key`, så flere navngivningsvarianter understøttes.

Når alle oplysninger er gyldige, anvendes den manuelle rute via `apply_manual_route()`, som samtidig håndterer metadata som `routeLabel`, `requestId` og `requestedBy`. Eventuel snapping til aktuel position eller OSRM-genberegning udføres ikke i `handle_message` men i de interne helper-funktioner, se bilag 5.e.

Funktionen logger både succes og fejl, så det kan spores præcist, hvilke manuelle ruteændringer simulatoren har modtaget og anvendt.

```

def handle_message(self, _client: mqtt.Client, _userdata, message: mqtt.MQTTMessage) -> None:
    topic = message.topic or ""
    if not self._matches_command_topic(topic):
        return

    data = self._decode_payload(message)
    if data is None:
        return

    command_type = str(data.get("type") or "").strip().lower()
    if command_type not in {"set_route", "assign_route", "update_route"}:
        return

    vehicle = self._resolve_vehicle(data)
    if not vehicle:
        return
    print(
        f"[command] Applying manual route to simulator vehicle {vehicle.vehicle_id} (plate {vehicle.number_plate}).",
        flush=True,
    )

    stops = self._parse_stops(data, vehicle.vehicle_id)
    if stops is None:
        return

    def read_key(*keys: str, default: str | None = None):
        for key in keys:
            if key in data and data[key] is not None:
                return data[key]
        return default

    base_speed = read_key("base_speed_kmh", "baseSpeedKmh")
    route_label = read_key("route_label", "routeLabel")
    request_id = read_key("request_id", "requestId")
    requested_by = read_key("requested_by", "requestedBy")
    try:
        vehicle.apply_manual_route(
            stops=stops,
            base_speed_kmh=float(base_speed) if base_speed is not None else None,
            route_label=route_label,
            request_id=request_id,
            requested_by=requested_by,
        )
        print(
            f"[command] Applied manual route to {vehicle.vehicle_id} "
            f"(request_id={request_id!r}, stops={len(stops)}",
            flush=True,
        )
    except Exception as exc:
        print(f"[command] Failed to apply route to {vehicle.vehicle_id}: {exc}", flush=True)

```

Figur 10.4: Modtagelse og behandling af manuelle ruteopdateringer via MQTT fra webapp → Web API → simulator.

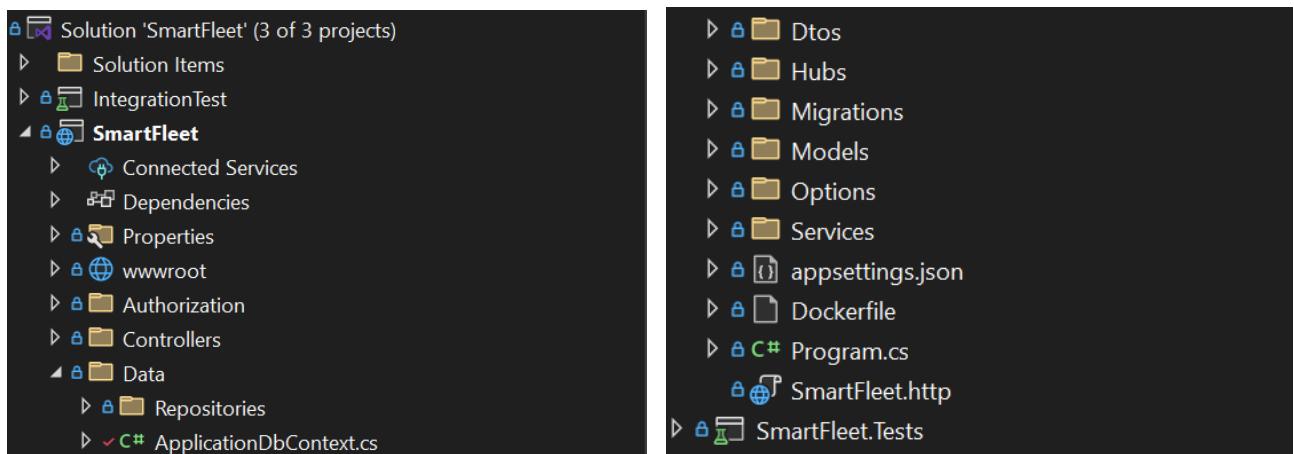
10.2 Web API implementering

I dette afsnit præsenteres den konkrete implementering af Web API’et. Fokus ligger på de centrale lag i løsningens trelagsarkitektur samt de væsentligste designvalg, som blev beskrevet i [9.3 Web API design](#).

Den fulde sourcekode til Web API’et findes i bilag [5.a](#).

10.2.1 Web API folder struktur

Som vist i figur 10.5 følger ASP.NET Core-projektet SmartFleet en trelagsstruktur bestående af controllers, services og repositories. Formålet med denne opdeling er at skabe tydelige ansvarsområder, reducere kobling og understøtte testbarhed på tværs af lagene.



(1) Projekt struktur for Web API'ets .NET solution

(2) Projekt struktur for Web API'ets .NET solution

Figur 10.5: Oversigt over Web API'ens ASP.NET Core solution. Strukturen afspejler den anvendte trelagsarkitektur, hvor præsentationslaget består af controllers, logiklaget af services og datalaget af repositories. Denne opdeling understøtter lav kobling, høj vedligeholdbarhed og en klar adskillelse af ansvar på tværs af Web API-projektet. Det ses yderligere at der er oprettet et unittest projekt og integrationstest projekt

Som beskrevet i afsnit 9.3 Web API design og illustreret i Web API-komponentdiagrammet figur 9.3, er systemet opbygget med en separation mellem præsentationslaget, logiklaget og infrastruktur/datalag. Denne struktur er omsat direkte til tre konkrete implementeringslag: *controllerlag*, *servicelag* og *repositorylag*. For en mere detaljeret gennemgang af disse lag, se bilag 5.d.

Denne opdeling afspejler også centrale aspekter af SOLID-principperne. *Single Responsibility Principle* ses ved at, at controllers udelukkende håndterer HTTP-kommunikation, services indeholder domænelogik, og repositories står for dataadgang. *Dependency Inversion Principle* understøttes ligeledes, idet controllers og services afhænger af interface-baserede abstraktioner, som injectes via dependency injection. På den måde kan konkrete datakilder og implementeringer udskiftes uden at de øvrige lag skal ændres.

10.2.2 Live telemetri

På Figur 10.6 og Figur 10.7 ses de to centrale funktioner, *PersistBatchAsync* og *BroadcastVehicleUpdateAsync*, som tilsammen udgør kernen i systemets behandling og udsendelse af live telemetridata. *PersistBatchAsync* håndterer selve persisteringen af indkomne data, mens *BroadcastVehicleUpdateAsync* står for real-time distribution af de opdaterede køretøjsinformationer til klienterne. Nedenfor gennemgås funktionernes roller mere detaljeret.

PersistBatchAsync

Når systemet modtager en batch af telemetribeskeder, behandler *PersistBatchAsync* dem én ad gangen. Funktionen opretter et DI-scope og henter de nødvendige repositories, hvorefter den forsøger at slå køretøjer op via *externalId* eller nummerplade. Findes køretøjet ikke i databasen, oprettes det. Dernæst opdateres køretøjets felter, herunder telemetridata, fører-relationer og relevante tidsstempler, og køretøjet tilføjes til en liste over alle enheder, der er blevet påvirket af batchen.

```
1 reference
private async Task PersistBatchAsync(IReadOnlyList<TelemetryEnvelope> batch, CancellationToken cancellationToken)
{
    if (batch.Count == 0)
    {
        return;
    }

    using var scope = _scopeFactory.CreateScope();
    var vehicleRepository = scope.ServiceProvider.GetRequiredService<IVehicleRepository>();
    var userRepository = scope.ServiceProvider.GetRequiredService<IUserRepository>();

    var byExternalId = new Dictionary<string, Vehicle>(StringComparer.OrdinalIgnoreCase);
    var byLicensePlate = new Dictionary<string, Vehicle>(StringComparer.OrdinalIgnoreCase);
    var vehiclesToBroadcast = new List<Vehicle>(batch.Count);

    foreach (var envelope in batch)
    {
        var vehicle = await UpsertVehicleAsync(
            vehicleRepository,
            userRepository,
            envelope,
            byExternalId,
            byLicensePlate,
            cancellationToken);

        vehiclesToBroadcast.Add(vehicle);
    }

    await vehicleRepository.SaveChangesAsync(cancellationToken);

    foreach (var vehicle in vehiclesToBroadcast.DistinctBy(v => v.Id))
    {
        await BroadcastVehicleUpdateAsync(vehicleRepository, vehicle, cancellationToken);
    }
}
```

Figur 10.6: Illustration af *PersistBatchAsync*-processen, hvor indkommende telemetribeskeder behandles samlet, køretøjsdata opdateres, og ændringer persisteres i databasen.

Når hele batchen er behandlet, gemmes alle ændringer samlet via `SaveChangesAsync()` for at sikre konsistens og minimere databasebelastning. Funktionen returnerer herefter en liste over de berørte køretøjer, som skal videre til real-time udsendelse.

Denne liste danner udgangspunktet for næste fase i processen, som illustreret i Figur 10.7, hvor køretøjsopdateringerne distribueres til systemets klienter.

```
1 reference
private async Task BroadcastVehicleUpdateAsync(IVehicleRepository vehicleRepository, Vehicle vehicle, CancellationToken cancellationToken)
{
    await vehicleRepository.LoadDriverAsync(vehicle, cancellationToken);

    try
    {
        var dto = vehicle.ToVehicleDto();
        var alertTimestamp = vehicle.LastTelemetryAtUtc.HasValue
            ? new DateTimeOffsetSet(DateTime.SpecifyKind(vehicle.LastTelemetryAtUtc.Value, DateTimeKind.Utc))
            : DateTimeOffset.UtcNow;
        await EmitAlertsAsync(vehicle, alertTimestamp, cancellationToken);
        NormalizeVehicleTelemetry(vehicle);
        await _vehicleHubContext.Clients.All.SendAsync(VehicleHub.VehicleUpdatedMethod, dto, cancellationToken);
    }
    catch (OperationCanceledException) when (cancellationToken.IsCancellationRequested)
    {
        // Ignore cancellations triggered by shutdown.
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, "Failed to broadcast telemetry update for vehicle {VehicleId}", vehicle.Id);
    }
}
```

Figur 10.7: Illustration af `BroadcastVehicleUpdateAsync`-processen, der udsender reeltidsopdateringer til klienterne via `SignalR`, efter at telemetridata er gemt.

BroadcastVehicleUpdateAsync

Som det fremgår af Figur 10.7, håndterer `BroadcastVehicleUpdateAsync` udsendelsen af opdaterede køretøjsdata efter persisteringen. For hvert berørt køretøj sikrer funktionen først, at den tilknyttede fører er korrekt indlæst. Derefter vurderes eventuelle forhold, der bør udløse en alarm, ved lav brændstofstand, offline-status eller hastighedsoverskridelser.

Når alle relevante oplysninger er samlet i en DTO, udsender funktionen en `VehicleUpdated`-besked via `SignalR VehicleHub`. Hvis der er identificeret alarmer, udsendes desuden en `VehicleAlert`. Dette sikrer, at alle klienter modtager liveopdateringer, og at både kortvisninger og lister i webappen opdateres øjeblikkeligt uden behov for reload.

10.2.3 Rute og køretøjsopdateringer

Funktionerne vist i Figur 10.8 og Figur 10.9 beskriver, hvordan Web API'et sender en ruteopdatering ud til et køretøj via MQTT.

Processen består af tre hovedtrin:

-
- 1) **Klargøring af ruteopdateringen.** Web API'et sikrer, at ruteopdateringen indeholder et gyldigt tidsstempel og er klar til afsendelse.
 - 2) **Forbindelse til en MQTT-broker.** Systemet forsøger at forbinde til den konfigurerede MQTT-broker og, hvis sat op, en fallback (fx localhost). Uden forbindelse sendes opdateringen ikke. Hvis brugernavn eller adgangskode er påkrævet, tilføjes dette automatisk. Uden en fungerende forbindelse kan opdateringen ikke sendes.

```
2 references
public async Task PublishRouteUpdateAsync(VehicleRouteCommandPayload payload, CancellationToken cancellationToken)
{
    var commandPayload = payload with
    {
        RequestedAtUtc = payload.RequestedAtUtc == default ? DateTime.UtcNow : payload.RequestedAtUtc,
    };

    var factory = new MqttFactory();
    using var client = factory.CreateMqttClient();

    var endpoints = ResolveBrokerEndpoints().ToArray();
    Exception? lastConnectError = null;

    foreach (var (host, port) in endpoints)
    {
        var clientOptionsBuilder = new MqttClientOptionsBuilder()
            .WithClientId($"vehicle-command-{Guid.NewGuid():N}".ToLowerInvariant())
            .WithCleanSession()
            .WithProtocolVersion(MqttProtocolVersion.V500);

        if (_options.UseWebSockets)
        {
            var endpoint = $"ws://:{host}:{port}";
            clientOptionsBuilder.WithWebSocketServer(options => options.WithUri(endpoint));
        }
        else
        {
            clientOptionsBuilder.WithTcpServer(host, port);
        }

        if (!string.IsNullOrWhiteSpace(_options.Username))
        {
            clientOptionsBuilder.WithCredentials(_options.Username, _options.Password ?? string.Empty);
        }

        var mqttClientOptions = clientOptionsBuilder.Build();
        await client.ConnectAsync(mqttClientOptions, cancellationToken);
        _logger.LogDebug("Connected to MQTT broker at {Host}:{Port}.", host, port);
        break;
    }
}
```

Figur 10.8: Proces over, hvordan Web API'et etablerer forbindelse til en MQTT-broker og publicerer en ruteopdatering til det relevante køretøj.

- 3) **Afsendelse af ruteopdateringen.** Når forbindelsen er etableret, bliver opdateringen sendt som en JSON-besked til et bestemt “topic”, som køretøjet lytter på. Beskeden sendes med en leveringsgaranti, der sikrer, at den når frem mindst én gang. Til sidst lukker Web API'et forbindelsen ned på en kontrolleret måde.

```

try
{
    var messagePayload = JsonSerializer.Serialize(commandPayload, _serializerOptions);
    var message = new MqttApplicationMessageBuilder()
        .WithTopic(_options.Topic)
        .WithPayload(messagePayload)
        .WithQualityOfServiceLevel(MqttQualityOfServiceLevel.AtLeastOnce)
        .WithRetainFlag(false)
        .Build();

    await client.PublishAsync(message, cancellationToken);
}
catch (Exception ex)
{
    _logger.LogError(ex, "Failed to publish vehicle route command for vehicle {VehicleId}.", payload.VehicleId);
    throw;
}
finally
{
    var disconnectOptions = new MqttClientDisconnectOptionsBuilder()
        .WithReason(MqttClientDisconnectOptionsReason.NormalDisconnection)
        .Build();
    await client.DisconnectAsync(disconnectOptions, cancellationToken);
}
}

```

Figur 10.9: Ruteopdateringen sendes som MQTT-besked til køretøjet

10.2.4 Chat 1:1

På Figur 10.10 ses funktionen *SendMessageAsync*, som håndterer afsendelsen af en 1:1-besked mellem to brugere. Funktionens ansvar spænder fra validering af indhold til oprettelse af selve beskeden og udsendelse af real-time notifikationer til begge deltagere. Nedenfor gennemgås dens funktionalitet mere detaljeret.

SendMessageAsync

Når en bruger sender en besked, validerer funktionen først, at beskedens indhold ikke er tomt. Herefter henter eller opretter den, den chat-tråd, som beskeden skal tilknyttes, via *GetOrCreateThreadAsync*. I den forbindelse sikres det, at både afsender og modtager er korrekt tilknyttet tråden.

Der oprettes derefter et nyt *ChatMessage*-objekt med status *Sent*, tidsstempel og reference til begge deltagere. Beskeden tilføjes repositorylaget, og chat-trådens *UpdatedAt*-felt opdateres for at afspejle den nye aktivitet. Når ændringerne er gemt i databasen, kalder funktionen notifier-komponenten, der via SignalR udsender en *ReceiveMessage*-begivenhed til begge brugere. Dette sikrer, at beskeden vises i realtid uden reload.

```

2 references
public async Task<ChatMessage> SendMessageAsync(int senderId, int recipientId, string body, CancellationToken ct)
{
    var (thread, _) = await GetOrCreateThreadAsync(senderId, recipientId, ct);
    var message = new ChatMessage
    {
        ThreadId = thread.Id,
        SenderId = senderId,
        RecipientId = recipientId,
        Body = body.Trim(),
        SentAt = DateTime.UtcNow,
        Status = ChatMessageStatus.Sent
    };

    await _chatRepository.AddMessageAsync(message, ct);

    thread.UpdatedAt = DateTime.UtcNow;

    await _chatRepository.SaveChangesAsync(ct);
    await BroadcastAsync(() => _notifier.MessageSentAsync(message, ct));
    return message;
}

```

Figur 10.10: Illustration af `SendMessageAsync`-processen, som håndterer oprettelse og afsendelse af chatbeskeder mellem brugere.

10.2.5 Authentication implementation

Systemet anvender JWT-baseret authentication med rollefordeling (Driver, Koordinator, Admin) samt OTP til oprettelse af brugere og nulstilling af adgangskoder. Løsningen består af fire hovedflows: registrering, login, OTP-håndtering og autorisation.

1. Registrering

Admin opretter brugere via `/api/auth/register`. Web API'et normaliserer e-mail, kontrollerer om brugeren allerede findes og opretter herefter brugeren med et midlertidigt password (`RequiresPasswordReset=true`). En OTP-mail udsendes via SendGrid, så brugeren kan fuldføre første login.

2. Login

Ved login via `/api/auth/login` verificerer Web API'et passwordet og udsteder derefter et JWT-token, som indeholder brugerens rolle og udløbstid. Tokenet anvendes både til API-kald og SignalR-hubs.

3. OTP og password reset

Brugere kan logge ind eller nulstille password ved hjælp af OTP via `/api/auth/login-otp` og `/api/auth/forgot-password`. OTP genereres i `OtpService` og lagres i cache og database. Når

OTP valideres, udstedes enten et JWT-token eller der gives adgang til at sætte et nyt password. Efter nulstilling fjernes reset-flagget.

4. Session- og adgangskontrol

UserSessionTracker sikrer, at hver bruger kun har én aktiv session. Autorisation håndteres via policies, som styrer adgang til adminpanel, ruteadministration, flådesiden og SignalR-hubs. API og hubs anvender samme JWT-validering.

SendGrid anvendes udelukkende til udsendelse af OTP- og reset-e-mails.

For at se en mere detaljeret beskrivelse af hele login flowet med kode, se bilag [5.g.](#)

10.3 Webapp implementering

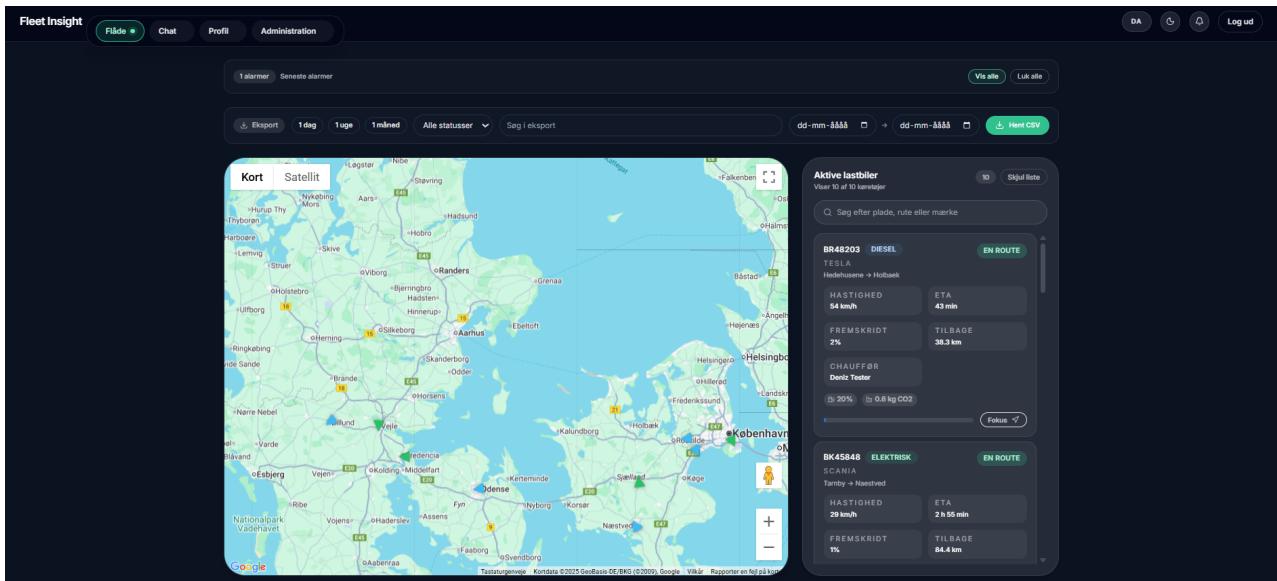
I dette afsnit bliver webapp implementeringen beskrevet. Webapp implementeringen bygger på de designvalg, der er beskrevet i afsnit [9.4 Webapp design](#). Webappen består af flere sider og funktioner, bl.a. login, flåde, profil, administration og chat. I dette afsnit er fokus dog på flådesiden, fordi det er her alle centrale krav fra user story 3 realiseres. Det omfatter live-telemetri, ruteændringer, kortvisning, alarmer og CSV-eksport — alle elementer, der udgør kernen i systemets reeltidsfunktionalitet. For at læse mere om CSV-eksport og funktionaliteterne hertil, se bilag [5.i.](#)

Flåde-siden kombinerer data fra Web API’et, liveopdateringer via SignalR og Google Maps og gør det muligt for brugeren at foreslå nye ruter.

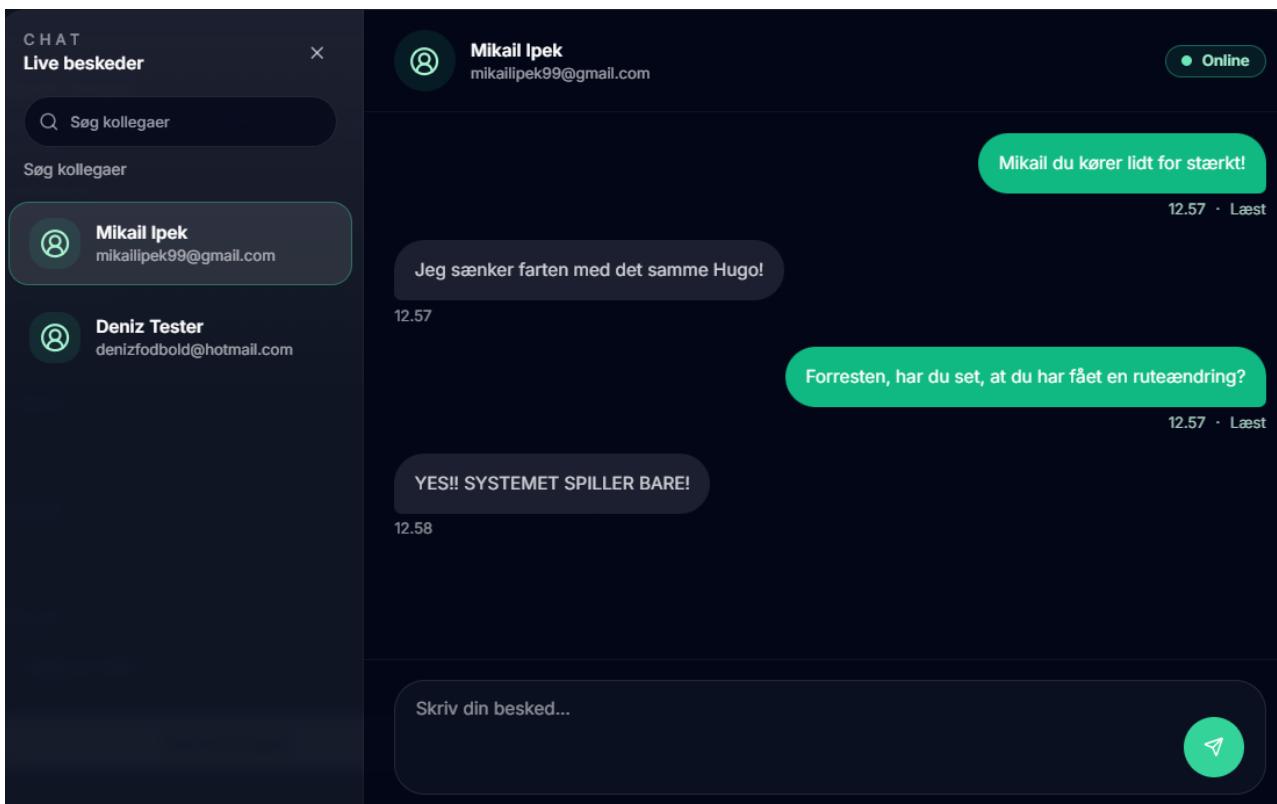
Den fulde sourcekode til webapp findes i bilag [5.b](#)

10.3.1 Resultatet af udviklingen

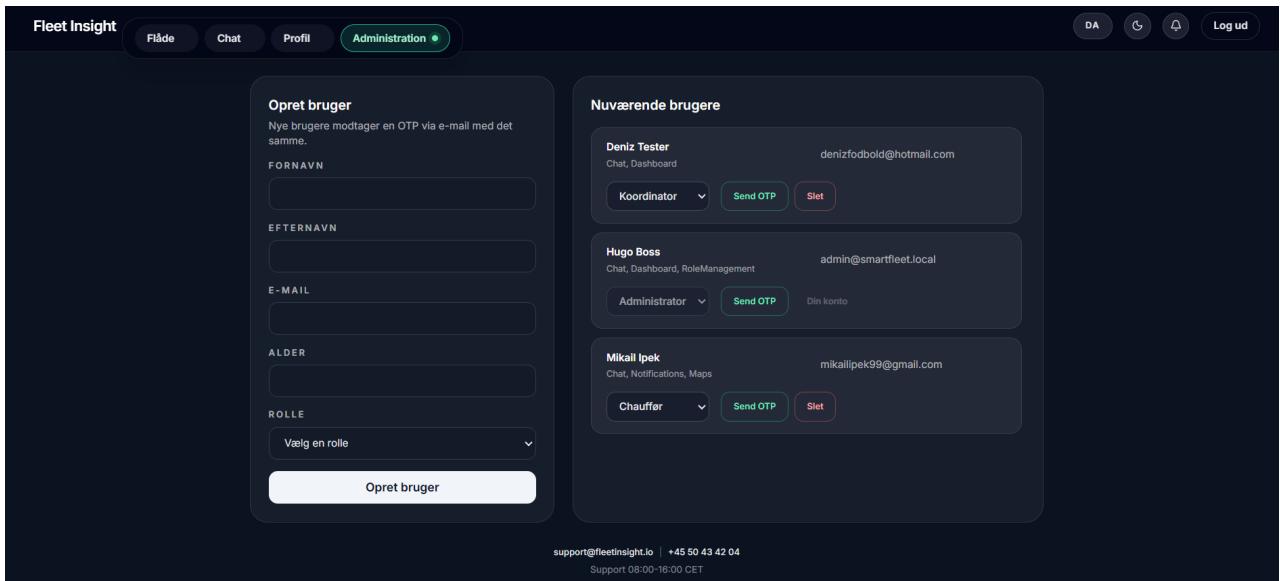
På figur [10.11](#), figur [10.12](#) og figur [10.13](#) ses de væsentligste færdigudviklede sider i Fleet Insight. De repræsenterer henholdsvis flådesiden, chatsiden og administrationspanelet—de tre centrale brugerflader, som tilsammen udgør hovedinteraktionen i systemet, og som har været fokus for rapporten. For at se en skitste af systemet før implementeringen, henvises til bilag [8.a Øvrige skærmbilleder](#) og alle webapp-sider kan ses samlet i bilag [5.f.](#)



Figur 10.11: Flådesiden, der viser kort, live-telemetri og detaljerede køretøjsoplysninger.



Figur 10.12: Chatsystemet med live beskeder, læsekwitteringer og brugerstatus.



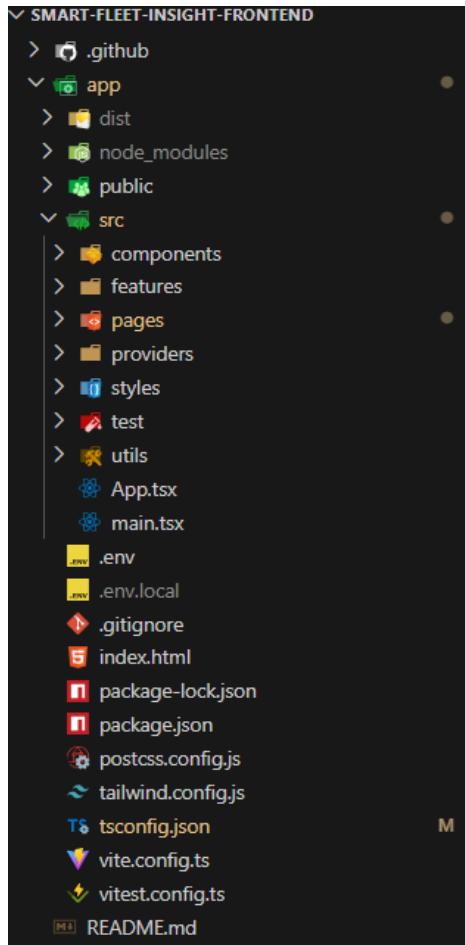
Figur 10.13: Administrationspanelet til oprettelse af brugere, ændring af roller og OTP-styring.

10.3.2 Webapp folder struktur

Figur 10.14 viser projektets mappestruktur, hvor koden er tydeligt opdelt efter ansvar:

- **pages**: indeholder applikationens sider, herunder flåde-siden.
- **features**: domænespecifik logik, fx hentning og håndtering af køretøjsdata.
- **providers**: globale tilstande som tema, sprog og authentication.
- **components**: mindre UI-komponenter, der kan bruges på tværs af sider.
- **utils**: hjælpefunktioner, fx beregning og formattering.

Opdelingen gør projektet let at vedligeholde og hjælper med at adskille datalogik, visning og systemintegrationer.



Figur 10.14: Webapp-projektets mappestruktur.

10.3.3 Hentning og normalisering af køretøjsdata

Figur 10.15 viser den kode, der henter og normaliserer køretøjsdata fra Web API’et. Da API’et ikke altid returnerer fuldt udfyldte eller konsistente felter, oversættes de rå data til en stabil og UI-venlig *Vehicle*-model, før de anvendes i webappen.

Normaliseringen sikrer robuste fallback-beregninger og ensartet håndtering af manglende værdier. Hvis `distanceRemainingM` mangler eller er ugyldig, beregnes den ud fra rutens længde og køretøjets progress som `routeDistanceKm * 1000 * (1 - progress)`. Tilsvarende udledes `kilometersDriven` fra `distanceTravelledM`, hvis den ikke er eksplicit angivet. Felter som eksternt id, karrosseritype og ruteopsummering konverteres til henholdsvis tom streng eller `null`, og førerdata kopieres kun, hvis feltet findes.

Formålet med denne normalisering er at stabilisere UI’et og sikre, at kort og oversigter fortsat fungerer korrekt, selv hvis API-data er ufuldstændige eller inkonsistente.

```

export function mapVehicle(vehicle: RawVehicle): Vehicle {
  const normalizedDistanceRemainingM =
    Number.isFinite(vehicle.distanceRemainingM) && vehicle.distanceRemainingM >= 0
      ? vehicle.distanceRemainingM
      : Number.isFinite(vehicle.routeDistanceKm) && Number.isFinite(vehicle.progress)
        ? Math.max(0, vehicle.routeDistanceKm * 1000 * (1 - vehicle.progress))
        : 0;

  const normalizedKilometersDriven =
    Number.isFinite(vehicle.kilometersDriven) && vehicle.kilometersDriven >= 0
      ? vehicle.kilometersDriven
      : Number.isFinite(vehicle.distanceTravelledM) && vehicle.distanceTravelledM >= 0
        ? vehicle.distanceTravelledM / 1000
        : 0;

  return {
    ...vehicle,
    distanceRemainingM: normalizedDistanceRemainingM,
    kilometersDriven: normalizedKilometersDriven,
    externalId: vehicle.externalId ?? '',
    bodyType: vehicle.bodyType ?? null,
    routeSummary: vehicle.routeSummary ?? null,
    driver: vehicle.driver
      ? {
          ...vehicle.driver,
        }
      : null,
  };
}

export async function getVehicles(): Promise<Vehicle[]> {
  try {
    const data = await http<RawVehicle[]>('/api/vehicles', { method: 'GET' });
    return data.map(mapVehicle);
  } catch (error) {
    if (error instanceof HttpError) {
      throw error;
    }
  }
}

```

Figur 10.15: Hentning og normalisering af køretøjsdata.

10.3.4 Manuel ruteændring fra kortet

Figur 10.16 viser den del af flådesiden, hvor en bruger manuelt kan foreslå en ny rute for et køretøj. Før indsendelse valideres stopnavnene, og ruten afvises, hvis et stop har et tomt eller ugyldigt navn. Stopnavnene trimmes, og der dannes en ruteopsummering, som anvendes som `routeLabel` og sendes til Web API’et sammen med stoplisten.

For at undgå genbrug af forældede rutedata ryddes den lokale rute-cache baseret på køretøjets cache-nøgle (`previousCacheKey`), inkl. eventuelle prefix-variante (`${previousCacheKey}:`). Hvis API’et returnerer et `requestId`, anvendes dette som rute-id; ellers faldes der tilbage til køretøjets eksisterende `routeId`. Der beregnes samtidig et entydigt ruteaftryk (`routeFingerprint`) til stabil identifikation.

Et midlertidigt rute-*override* oprettes herefter in-memory med estimeret distance og ETA

samt tidsstempler for gyldighed (`appliedAt` og `expiresAt`). Ruteopsummeringen normaliseres (`normalizeRouteSummary`) før den gemmes i override'et for at sikre konsistent repræsentation. Override'et persisteres desuden til `localStorage` via `persistRouteOverrides()`, så den optimistiske UI-tilstand kan genskabes ved genindlæsning.

```
const unnamedStop = filledStops.find((stop) => stop.name.trim().length === 0);
if (unnamedStop) {
  toast.error(t('maps.routeEditor.validation.missingName'));
  return;
}

setSubmittingRoute(true);
try {
  const stopsPayload = filledStops.map((stop) => ({
    name: stop.name.trim(),
    latitude: stop.latitude,
    longitude: stop.longitude,
  }));
  const previousCacheKey = getRouteCacheKey(selectedVehicle);
  Array.from(directionsCache.current.keys()).forEach((key) => {
    if (key === previousCacheKey || key.startsWith(`${previousCacheKey}:`)) {
      directionsCache.current.delete(key);
    }
  });
  const stopNames = stopsPayload.map((stop) => stop.name);
  const optimisticSummary = stopNames.join(' -> ');
  const routeLabel = optimisticSummary.length > 0 ? optimisticSummary : undefined;
  const response = await updateVehicleRoute(selectedVehicle.id, {
    stops: stopsPayload,
    routeLabel,
  });
  const nextRouteIdRaw = response.requestId ? response.requestId.trim() : selectedVehicle.routeId;
  const nextRouteId = nextRouteIdRaw && nextRouteIdRaw.length > 0 ? nextRouteIdRaw : selectedVehicle.routeId;
  const nextRouteIdFingerprint = toRouteIdentifier(nextRouteId);
  directionsCache.current.delete(previousCacheKey);
  const estimatedDistanceKm = routeDraftSummary?.distanceKm ?? null;
  const estimatedEtaSeconds = routeDraftSummary ? Math.round(routeDraftSummary.durationMinutes * 60) : null;
  const normalizedSummary = normalizeRouteSummary(optimisticSummary);

  const selectedFingerprint = getVehicleFingerprint(selectedVehicle);
  pendingRouteOverridesRef.current.set(selectedFingerprint, {
    summary: optimisticSummary,
    normalizedSummary,
    routeId: nextRouteId ?? null,
    routeFingerprint: nextRouteIdFingerprint,
    distanceKm: estimatedDistanceKm,
    etaSeconds: estimatedEtaSeconds,
    expiresAt: Date.now() + ROUTE_OVERRIDE_TTL_MS,
    appliedAt: Date.now(),
    stops: stopsPayload,
  });
  persistRouteOverrides();
  lastRoutedVehicleKey.current = null;
}
```

Figur 10.16: Flow for manuel ruteændring i webappens ruteeditor.

10.4 Infrastruktur og deployment

I dette afsnit gennemgås de elementer, der ligger ved siden af selve softwareudviklingen, herunder de overvejelser der normalt vedrører infrastruktur og build-processer. Da projektet ikke omfatter udrulning til egentlige miljøer, fokuserer afsnittet på den etablerede CI-pipeline og build-proces.

10.4.1 Hosting

Som vist på Figur 10.17 hostes databasen i Neon, hvilket sikrer, at systemet altid har adgang til en stabil og online datakilde. Dette er vigtigt for, at realtidsfunktionerne kan fungere uden afbrydelser. Hvis databasen kun lå lokalt, ville de forskellige services ikke kunne nå den, og hele realtidsflowet ville bryde sammen. Hosting i Neon har derfor været en praktisk nødvendighed for at få systemet til at hænge sammen og køre i realtid.

The screenshot shows the NeonDB interface with the 'Smartfleet' project selected. On the left, there's a sidebar with 'PROJECT' and 'BRANCH' sections. Under 'PROJECT', 'Dashboard', 'Branches', 'Integrations', 'Auth', and 'Settings' are listed. Under 'BRANCH', 'production' is selected, showing 'Overview', 'Monitoring', 'SQL Editor', 'Tables' (which is highlighted), 'Backup & Restore', and 'Data Masking (Beta)'. Under 'APP BACKEND', 'Data API' and 'Feedback' are listed. At the bottom of the sidebar is a 'Collapse menu' button. The main area is titled 'Tables' and shows a list of databases: 'neondb' (selected), 'Database studio', 'public', and '_EFMigrationsHistory'. Under 'public', there are tables: 'ChatMessages', 'ChatThreads', 'Users', and 'Vehicles' (which is also highlighted). The 'Vehicles' table is displayed as a grid with columns: Id (integer), UpdatedAt (timestamp with time zone), VehicleType (varchar(100)), FuelType (varchar(50)), and BodyType (varchar(50)). The table contains 21 rows of vehicle data. The top right of the interface shows 'All ok', a refresh icon, and a user icon. Below the table are buttons for 'Add record', 'Filters', 'Columns', and 'Add column'. The bottom right shows pagination with '50 rows • 172ms' and icons for navigating through the data.

Id	UpdatedAt	VehicleType	FuelType	BodyType
1	2025-10-03 09:38:06.34...	string	string	string
2	2025-12-05 13:54:50.62...	Scania	diesel	
3	2025-12-05 13:54:50.73...	Volvo	diesel	
4	2025-12-05 13:54:50.83...	Mercedes	diesel	
5	2025-12-05 13:54:50.93...	Volvo	diesel	
6	2025-12-05 13:54:51.04...	Volvo	diesel	
7	2025-12-05 13:54:51.14...	Tesla	electric	
8	2025-12-05 13:54:51.14...	MAN	electric	
9	2025-12-05 13:54:51.24...	Scania	electric	
10	2025-12-05 13:54:51.24...	Scania	electric	
11	2025-12-05 13:54:51.35...	Volvo	diesel	
12	2025-11-15 23:05:49.49...	MAN	electric	
13	2025-11-15 23:05:49.59...	MAN	diesel	
14	2025-11-15 23:05:49.69...	Volvo	electric	
15	2025-11-15 23:05:49.79...	BMC	electric	
16	2025-11-15 23:05:49.89...	Mercedes	diesel	
17	2025-11-15 23:05:50.03...	MAN	diesel	
18	2025-11-15 23:05:50.22...	Volvo	electric	
19	2025-11-15 23:05:50.42...	MAN	diesel	
20	2025-11-15 23:05:50.61...	Tesla	diesel	
21	2025-11-15 23:05:50.80...	Tesla	diesel	

Figur 10.17: Oversigt over databasens tabeller og indhold i NeonDB, som anvendes til hosting af systemets centrale data (vehicles, users, chat og telemetrydata)

10.4.2 CI pipeline

Projektet bruger CI til automatisk at integrere kodeændringer og håndtere build- og testprocessen. Der er oprettet tre pipelines, som hver især tester simulatoren, Web API'et og webappen. Web API-pipelinen bygger desuden API-projektet. Pipelinen kompilerer projekterne og kører unittests for at sikre, at ændringer ikke introducerer fejl. CI-pipelinen for simulatoren og webappen kan ses i bilag 5.h.

Pipelines aktiveres automatisk, når der laves ændringer i master-branch eller når der oprettes et PR. Figur 10.18 viser kørselshistorikken for Web API-pipelinen, og figur 10.19 viser et eksempel på en fuldført afvikling af samme pipeline.

All workflows			
Showing runs from all workflows			
20 workflow runs			
Event	Status	Branch	Actor
Fixed unittest (#35)	main	12 minutes ago	...
.NET Tests #20: Commit d8f6c85 pushed by Deniz509		39s	
Fixed unittest	Fix-unittest	13 minutes ago	...
.NET Tests #19: Pull request #35 opened by Deniz509		1m 20s	
Fixed unittest	Fix-unittest	14 minutes ago	...
.NET Tests #18: Commit ed0f6ce pushed by Deniz509		40s	
fixed sengrid (#34)	main	Dec 5, 12:56 PM GMT+1	...
.NET Tests #17: Commit 68c6688 pushed by Deniz509		53s	
fixed sendgrid	Fix-sendgrid-configuration	Dec 5, 12:55 PM GMT+1	...
.NET Tests #16: Pull request #34 opened by Deniz509		36s	
fixed sengrid	Fix-sendgrid-configuration	Dec 5, 12:54 PM GMT+1	...
.NET Tests #15: Commit 569b0d0 pushed by Deniz509		1m 9s	

Figur 10.18: Oversigt over tidligere kørsler af Web API-buildpipen, som afvikles automatisk for at bygge testprojektet og afvikle unittests ved både commits og pull requests.

> <input checked="" type="checkbox"/> Set up job	1s
> <input checked="" type="checkbox"/> Checkout	1s
> <input checked="" type="checkbox"/> Setup .NET SDK	1s
> <input checked="" type="checkbox"/> Restore dependencies	11s
> <input checked="" type="checkbox"/> Build	13s
> <input checked="" type="checkbox"/> Run tests	6s
> <input checked="" type="checkbox"/> Post Setup .NET SDK	0s
> <input checked="" type="checkbox"/> Post Checkout	0s
> <input checked="" type="checkbox"/> Complete job	0s

Figur 10.19: Oversigt over, hvordan Web API-pipelinen bygger og tester projektet gennem en række automatiserede trin.

11 Test

Dette afsnit præsenterer testarbejdet for systemets tre hovedkomponenter: simulatoren, Web API'et og webappen. For simulatoren og webappen er der udelukkende gennemført unit tests, mens Web API'et er testet både med unit tests og udvalgte integrationstests.

Integrationstests for simulatoren og webappen er fravalgt, da disse komponenter er stærkt afhængige af eksterne systemer såsom MQTT, OSRM, Google Maps API, SignalR-hubs og Web API'et. Et fuldt automatiseret testsetup ville kræve omfattende containeropsætning og orkestrering af flere services og vurderes derfor ikke effektivt inden for projektets tidsramme. I stedet er integrationerne verificeret manuelt i et kørende end-to-end setup, mens den interne logik er dækket af omfattende unit tests.

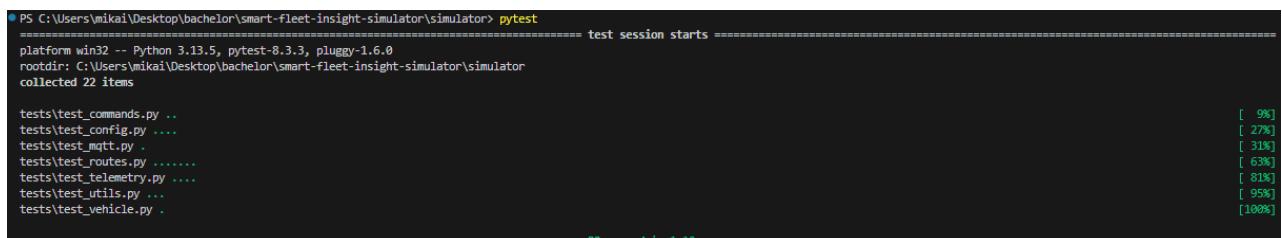
11.1 Test af simulatoren

Simulatoren er testet med fokus på de centrale, logiske komponenter: kommandobehandling, rutealgoritmer, telemetriproduktion, MQTT-publisheren samt hjælpefunktioner til distance- og tidsberegning. Testene er skrevet som unittests med `pytest`, og der er udarbejdet tests for de mest kritiske funktioner, så fejl i rutehåndtering, telemetri og publicering fanges tidligt.

11.1.1 Unit tests

På Figur 11.1 ses resultatet af en testkørsel, hvor alle unittests er kørt gennem `pytest`. I alt 22 tests blev kørt og alle bestod uden fejl.

Unit testsene sikrer dermed, at simulatorens kernefunktioner arbejder korrekt isoleret — uden at Web API'et, OSRM eller MQTT behøver at være startet.



```
PS C:\Users\mikai\Desktop\bachelor\smart-fleet-insight-simulator\simulator> pytest
===== test session starts =====
platform win32 -- Python 3.13.5, pytest-8.3.3, pluggy-1.6.0
rootdir: C:\Users\mikai\Desktop\bachelor\smart-fleet-insight-simulator\simulator
collected 22 items

tests\test_commands.py ...
tests\test_config.py .....
tests\test_mqtt.py .
tests\test_routes.py .....
tests\test_telemetry.py .....
tests\test_utils.py ...
tests\test_vehicle.py .

[  0%]
[ 27%]
[ 31%]
[ 63%]
[ 81%]
[ 95%]
[100%]

22 passed in 1.12s
```

Figur 11.1: Resultatet af `pytest`-kørsel for simulatorens enhedstest. Alle 22 tests er gennemført med succes.

11.1.2 Smoke-test

Simulatoren blev smoke-testet ved hjælp af en midlertidig webapp placeret i samme mappe som simulatoren. Denne løsning gjorde det muligt at køre simulatoren stand-alone, uden forbindelse til Web API’et, for at verificere at telemetri, ruteopdateringer og generelt dataflow blev genereret korrekt og kunne vises i webappen. Formålet var at sikre, at simulatorens grundfunktionalitet fungerede som forventet, før den blev integreret i resten af systemet.

For at se en mere detaljeret beskrivelse af dette, se bilag [7.b](#)

11.2 Test af Web API

Der er udarbejdet både unit tests og integrationstests til Web API’et. Unit tests (xUnit) benytter mocks og EF InMemory til isoleret test af adfærd ved hjælp af FluentAssertions og Moq. Integrationstests (xUnit) kører mod de faktiske services med rigtige repositories og EF InMemory som database for at validere samspillet mellem service- og datalag.

11.2.1 Unit test

Figur [11.2](#) viser resultatet af en unittest-kørsel. Testene omfatter blandt andet OTP- og JWT-håndtering, sessiontracking, autentificeringsflowet, chatfunktionalitet, validering i VehicleCommandPublisher samt udvalgte telemetry-funktioner.

ZOMBIE-princippet har været anvendt som udgangspunkt, især i forhold til Zero/One/Many-scenerier og test af fejlveje i Auth-, Chat- og CommandPublisher-logikken. Princippet er ikke fuldt implementeret i alle metoder, men har guidet opbygningen af de centrale testcases.

Test run finished: 53 Tests (53 Passed, 0 Failed, 0 Skipped) run in 1,7 sec		
Test	Duration	Traits
SmartFleet.Tests (32)	2,7 sec	
SmartFleet.Tests.Services (32)	2,7 sec	
VehicleCommandPublisherTests (6)	134 ms	
UserSessionTrackerTests (7)	624 ms	
TelemetryHelpersTests (2)	115 ms	
OtpServiceTests (6)	732 ms	
JwtTokenServiceTests (2)	119 ms	
ChatServiceTests (3)	822 ms	
AuthServiceTests (6)	163 ms	

Figur 11.2: Resultatet af en unittest-kørsel, hvor alle tests er gennemført med succes.

Figur 11.3 viser en testkørsel med fuld line-coverage for de testbare dele af servicelaget. Coverage-procenten angiver, hvor stor en del af kodebasen der er dækket af testene.

Name	Covered	Uncovered	Coverable	Total	Line coverage
- SmartFleet.Tests	512	0	512	826	100%
SmartFleet.Tests.Services.AuthServiceTests	96	0	96	159	100%
SmartFleet.Tests.Services.ChatServiceTests	58	0	58	94	100%
SmartFleet.Tests.Services.JwtTokenServiceTests	59	0	59	87	100%
SmartFleet.Tests.Services.OtpServiceTests	134	0	134	203	100%
SmartFleet.Tests.Services.TelemetryHelpersTests	42	0	42	70	100%
SmartFleet.Tests.Services.UserSessionTrackerTests	62	0	62	108	100%
SmartFleet.Tests.Services.VehicleCommandPublisherTests	61	0	61	105	100%

Figur 11.3: Coverage-oversigt for unittests i servicelaget.

11.2.2 Integrationstest

Figur 11.4 viser resultaterne af integrationstestene.

Name	Covered	Uncovered	Coverable	Total	Line coverage	Covered	Total	Branch coverage
+ IntegrationTest	264	0	264	402	100%	42	42	100%

Figur 11.4: Resultater af gennemførte integrationstests.

Integrationstestene fokuserer på *UserService* og validerer servicen i samspil med de faktiske repositories og EF InMemory som database. Testene kører ikke via controllers eller HTTP og inddrager heller ikke eksterne systemer som PostgreSQL, Auth, Chat, MQTT eller SignalR.

De centrale scenarier omfatter oprettelse, hentning, opdatering og sletning af brugere samt håndtering af password, roller og profiloplysninger. Formålet er at sikre, at *UserService* og repositorylaget fungerer korrekt som en samlet enhed, og at data persisteres og hentes som forventet.

Der er valgt udelukkende at teste *UserService* på integrationsniveau, da dette område muliggør et stabilt og realistisk testsetup uden behov for eksterne systemer. De øvrige dele af systemet kræver kørende eksterne komponenter, hvilket ville kræve et langt mere omfattende testmiljø og ligger uden for projektets rammer.

11.2.3 Smoke-test

Der blev udført manuelle smoke tests af Web API'et via Swagger, hvor hvert API-endpoint blev testet isoleret for at verificere korrekte HTTP-statuskoder, svarformater og fejlmeddelelser,

inden integrationen med webappen. Dette gav en hurtig sanity-check af de centrale funktioner. Testen fungerer dog ikke som en automatiseret regressionstest og omfatter heller ikke realtidskomponenterne, som valideres separat.

For at se en mere detaljeret beskrivelse om dette, se bilag [7.c](#).

11.3 Test af webapp

Webappens testarbejde er fokuseret på de dele af frontend-koden, der kan testes isoleret uden afhængigheder til eksterne systemer. Formålet er at sikre korrekt og forudsigelig adfærd i den klient-side logik, mens UI-tunge funktioner og integrationer verificeres manuelt i et kørende setup med Web API’et og simulatoren. Testene er implementeret som unittests ved hjælp af *Vitest*.

11.3.1 Unit tests

Som vist i Figur 11.5 er der udarbejdet unittests for webappens centrale hjælpefunktioner og databehandling. I alt blev otte testfiler med 33 tests kørt med succes. Testene dækker blandt andet normalisering og forberedelse af data og sikrer, at webappen håndterer mangelfulde eller inkonsistente API-responser robust.

```
PS C:\Users\mikai\Desktop\bachelor\smart-fleet-insight-frontend\app> npm test

> app@0.0.0 test
> vitest run

RUN v2.1.9 C:/Users/mikai/Desktop/bachelor/smart-fleet-insight-frontend/app

✓ src/utils/authStorage.test.ts (3)
✓ src/utils/http.test.ts (2)
✓ src/utils/passwordStrength.test.ts (3)
✓ src/utils/url.test.ts (4)
✓ src/pages/maps/MapsPage.test.ts (3)
✓ src/pages/maps/mapUtils.test.ts (12)
✓ src/features/vehicles/api/getVehicles.test.ts (4)
✓ src/features/vehicles/api/updateVehicleRoute.test.ts (2)

Test Files 8 passed (8)
Tests 33 passed (33)
Start at 11:10:21
Duration 2.23s (transform 820ms, setup 93ms, collect 1.83s, tests 57ms, environment 2ms, prepare 1.56s)
```

Figur 11.5: Resultat af Vitest-kørsel for webappen. Alle 33 tests passeret.

11.4 Integration mellem webapp og Web API

I de tidligere afsnit er det beskrevet, hvordan Web API'et er testet med unit tests og integrationstests, hvilket sikrer kvaliteten af backendens enkelte dele. For at teste sammenspillet mellem webapp og Web API er der derefter udført manuelle end-to-end tests.

Webappen er her blevet koblet til en kørende version af API'et, hvorefter centrale funktioner er testet i praksis. Testene inkluderede også realtidskommunikation.

Formålet har været at sikre, at handlinger i webappen udløser de forventede ændringer i API'et og databasen. Disse tests er gennemført løbende i slutningen af hver iteration, da der ikke findes en fuldt automatiseret testpipeline, der dækker hele systemet.

12 Accepttest

Dette afsnit beskriver de accepttests, der er blevet gennemført under udviklingen af produktet. Der er blevet udført en stor accepttest, hvor alle kravene for Fleet Insight er blevet testet til sidst i processen.

Accepttesten er udarbejdet og gennemført internt i projektgruppen. Testen tager udgangspunkt i systemkravene beskrevet i afsnit [5 Kravspecifikation](#) og fokuserer på at verificere, at systemet fungerer som forventet.

12.1 Resultater af accepttest

Accepttesten blev afholdt d. 26/11/2025, internt af alle gruppemedlemmer. Udførslen af accepttesten er blevet udført af en i gruppen, hvor de andre to har observeret om det er accepteret eller ej. I accepttesten blev følgende accepteret:

Alle user stories blev accepteret.

16/16 **must** krav blev accepteret

3/3 **should** krav blev accepteret

2/5 **could** krav blev accepteret

2/2 **won't** krav blev accepteret

Da user stories 2, 3 og 5 dækker de væsentligste funktionelle kerneområder, live flådeoverblik, chat og brugeradministration, er det besluttet at præsentere accepttesten for disse tre som repræsentative eksempler. Den fulde accepttest kan ses i bilag [7.a](#).

Ved udarbejdelsen af accepttesten er hvert scenarie i user stories blevet omsat til konkrete funktionelle krav. Dette sikrer, at alle dele af user storyen - herunder prækonditioner, handlinger og forventede resultater - testes struktureret og entydigt. I tabellerne er kravene opdelt i selvstændige testtrin, så hvert scenarie kan verificeres enkeltvis, hvilket giver et solidt grundlag for at afgøre, om user story'en samlet set er opfyldt.

Tabel [12.1](#), [12.2](#) og [12.3](#) viser accepttesten og resultaterne for User Story 2, 3 og 5. Samtlige testtrin er bestået, og disse user stories er derfor accepteret.

I tabel [12.4](#) ses accepttestene samt resultaterne for alle ikke-funktionelle **must**-krav. Det kan ses, at alle disse krav er blevet accepteret.

Krav	Prioritet	Testspecificering	Forventet resultat	Resultat
F.1	Must	Prækondition: Admin er logget ind. 1) Gå til opret-bruger-siden. 2) Indtast e-mail, navn, alder og rolle. 3) Klik “Opret bruger”.	Brugeren oprettes, og systemet sender en bekræftelsesmail til brugeren.	OK
F.2	Must	Prækondition: Admin er logget ind. 1) Gå til opret-bruger-siden. 2) Lad et obligatorisk felt stå tomt. 3) Klik “Opret bruger”.	Systemet afviser oprettelsen og viser fejlmeldingen “Dette felt er påkrævet”.	OK

Tabel 12.1: Accepttest for user story 2, som verificerer oprettelse af brugere. Testen er udført med en admin-bruger, der allerede er logget ind.

Accepttesten for user story 3 indeholder ikke fejlscenarierne. For at se den fulde accepttest for user story 3, se bilag 7.a

Krav	Prioritet	Testspecificering	Forventet resultat	Resultat
F.1	Must	Prækondition: Bruger med adgang er logget ind og på siden “Flåde”. <ol style="list-style-type: none"> 1) Åbn (eller forbliv på) ”Flåde”. 2) Observer kort og listevisning. 	Kortet viser alle køretøjer; liste viser relevant telemetri. Positioner og data opdateres løbende uden side-refresh.	OK
F.2	Must	Prækondition: Bruger med adgang er logget ind og på ”Flåde”. <ol style="list-style-type: none"> 1) Klik ”Fokus” på et køretøj. 2) Klik ”Rediger rute”. 3) Indtast ny destination. 4) Klik ”Opslag”. 5) Klik ”Send rute”. 	Ruten opdateres; chaufføren modtager notifikation og kan se den opdaterede rute.	OK
F.3	Must	Prækondition: Bruger med adgang er logget ind og på ”Flåde”. <ol style="list-style-type: none"> 1) Vælg dato eller interval. 2) Vælg statusser. 3) Klik ”Hent CSV”. 	CSV downloades med korrekt filtreret data.	OK
F.4	Must	Prækondition: Koordinator er logget ind på ”Flåde”. <ol style="list-style-type: none"> 1) Udløs hændelse: for høj hastighed, lavt brændstof eller ingen data i 5 min. 2) Observer alarm. 3) Åbn og luk alarm. 	Koordinator modtager korrekt alarm; alarmer vises med detaljer og kan lukkes.	OK

Tabel 12.2: Accepttest for user story 3, som omhandler visning af live køretøjsdata, ruteændring, dataeksport og alarmer. Testen er udført med en bruger med adgang, der allerede er logget ind på siden ”Flåde”. Tabellen viser de funktionelle krav, testspecifikationer og resultater.

Krav	Prioritet	Testspecificering	Forventet resultat	Resultat
F.1	Must	Prækondition: Bruger er logget ind og vil kommunikere med en kollega. 1) Klik "Chat" (chat-modal åbnes). 2) Klik på en kollega fra listen. 3) Systemet indlæser seneste beskeder. 4) Skriv en besked og klik "Send".	Beskeden vises med status "Sendt"; kollegaen modtager notification og beskeden i sin chat.	OK
F.2	Must	Prækondition: Bruger er logget ind og vil kommunikere med en kollega. 1) Åbn chatten for en kollega. 2) Skriv en besked og klik "Send".	Beskeden sendes ikke; systemet viser "Kunne ikke sende beskeden. Prøv igen." Ingen ny besked oprettes i databasen.	OK

Tabel 12.3: Accepttest for user story 5, som tester 1:1-chatfunktionen mellem brugere. Testen er gennemført med en bruger, der allerede er logget ind, og billedet viser de funktionelle krav, testspecifikationer og resultater.

Krav	Prioritet	Testspecificering	Forventet resultat	Resultat
IF.1	Must	Prækondition: Brugere findes med rollerne Driver, Koordinator, Admin. 1) Login som Driver og forsøg at åbne admin-funktioner. 2) Login som Admin og åbn admin-funktioner.	Adgang styres af rolle; Admin har adgang.	OK
IF.2	Must	Prækondition: Live-kort med aktive køretøjer. 1) Åbn live-kort. 2) Verificér live-opdatering 10 sek. 3) Brug søgning/filtre.	Kort og liste opdateres live; søg/filtrér fungerer korrekt.	OK
IF.3	Must	Prækondition: To brugere er online. 1) Send besked. 2) Tjek sendt/læst-status. 3) Tjek modtagerens notifikation. 4) Simulér fejl ved afsendelse.	Chat viser sendt/læst; fejl håndteres korrekt.	OK
IF.4	Must	Prækondition: API eller hub er utilgængelig. 1) Udløs handling der kalder API/hub. 2) Observer UI-respons.	Bruger-venlig fejlmeddeelse; ingen nedbrud.	OK
IF.5	Must	1) Mål loadtid for flådeside og live-kort.	Flådeside og live-kort loader 2 sek.	OK
IF.6	Must	1) Inspicer databasen. 2) Bekræft hashed passwords. 3) Valider korrekt lagring af e-mail og roller.	Ingen klartekst-passwords; data håndteres sikkert.	OK
IF.7	Must	Ruteopdatering 1) Admin/Koordinator sender ny rute. 2) Mål leveringstid til chauffør.	Ny rute vises hos chauffør 10 sek uden reload.	OK
IF.8	Must	1) Vælg dato/status. 2) Klik "Hent CSV".	CSV downloades korrekt med filtreret data.	OK
IF.9	Must	1) Opdater navn/e-mail/adgangskode. 2) Forsøg ugyldige ændringer.	Gyldige ændringer gemmes; meningsfulde fejl vises.	OK

Tabel 12.4: Accepttestresultater for de ikke-funktionelle must-krav. Billedet viser de krav, der vurderes absolut nødvendige for systemets drift, herunder sikkerhed, performance, datasikkerhed, live-opdatering og fejhåndtering. Alle must-krav er testet og bestået.

13 Diskussion

Problemformuleringen lyder:

Hvordan kan man give flådekoordinatører og chauffører et samlet, realtidsopdateret overblik over køretøjsstatus, ruter og beskeder, så de kan træffe hurtige beslutninger og reagere rettidigt på hændelser?

Fleet Insight adresserer de centrale udfordringer beskrevet i afsnit [4.1 Problem beskrivelse](#) ved at samle telemetri, ruter, alarmer og kommunikation i én platform. Prototypen demonstrerer, at et samlet realtidsbillede kan etableres og fungere som beslutningsstøtte, når data behandles og distribueres gennem en fælles arkitektur. Resultaterne skal dog ses i lyset af projektets afgrænsninger: Telemetri og belastning er kun testet i et simuleret miljø, og systemet er ikke afprøvet i realistiske driftsforhold. Afsnittet diskuterer derfor både styrker, begrænsninger og teknologiske valg, samt hvilke konsekvenser disse har for projektets resultat.

13.1 Resultater

Accepttesten viste ingen væsentlige problemer. Alle **must**- og **should**-krav blev opfyldt, to **could**-krav blev accepteret, og ydermere blev alle **won't**-krav accepteret. Dette indikerer, at projektet har fulgt de definerede afgrænsninger og prioriteringer i kravspecifikationen.

13.2 Styrker og begrænsninger

Udviklingen af Fleet Insight har fremhævet både tydelige styrker og væsentlige udfordringer. En af systemets største styrker er det stabile realtidsflow mellem simulator, Web API og webapp. Kombinationen af MQTT til telemetri og SignalR til liveopdateringer har muliggjort et smidigt dataflow uden manuel refresh, og accepttestene viste, at ruteændringer blev vist inden for de forventede 10 sekunder. Simulatoren var også en markant fordel, da den gjorde det muligt at udvikle og teste uden fysisk udstyr. C4-modellen bidrog til en klar, konsistent arkitektur og gjorde det lettere at bevare overblikket over systemets struktur.

Samtidig blev der identificeret begrænsninger, som påvirker systemets modenhed. Simuleret telemetri afspejler ikke den variation, netværksstøj og uregelmæssighed, der kendetegner virkelige flåder. Det er derfor uklart, hvilke dataformater, protokoller og opdateringsfrekvenser fysisk hardware anvender, og hvordan dette påvirker performance. Integration med reel telemetrihardware blev ikke undersøgt dybdegående.

Skalerbarhedstests blev kun gennemført på personlige computere, og webappen udviste stabilitetsproblemer ved omkring 100 samtidige køretøjer. Årsagen til dette blev ikke analyseret nærmere, og systemet blev ikke testet på hardware med større ressourcer. Det er derfor usikkert, hvor godt løsningen vil håndtere belastning i en realistisk driftskontekst.

Samlet set viser vurderingen, at Fleet Insight teknisk set kan etablere et samlet realtidsbillede af flåden, men at praktisk anvendelse i produktion kræver integration med fysisk hardware, forbedret skalerbarhed og generel robusthed.

13.3 Refleksion over teknologivalg

Valget af MQTT har været hensigtsmæssigt for en prototype, da protokollen er letvægts og velegnet til hyppige opdateringer. I større produktionssystemer kan manglen på kø-håndtering og persistens dog være en udfordring. SignalR leverede stabile realtidsopdateringer, men er afhængig af WebSockets og dermed følsomt over for hostingmiljøets stabilitet. OSRM fungerede som en effektiv ruteplanlægger i udviklingsfasen, men open-source kortdata kan afvige fra virkeligheden, og kommercielle tjenester vil ofte tilbyde hurtigere og mere præcise ruteplanlægningsevner.

13.4 Perspektivering

Fleet Insight har potentiale til at udvikle sig til et fuldt flådestyringssystem, og flere funktioner peger allerede mod praktiske, værdiskabende anvendelser. En væsentlig fremtidig mulighed er eksport af historiske telemetridata i CSV-format, suppleret med beregninger af eksempelvis energiforbrug eller CO₂-udledning. Dette kan understøtte virksomheders rapportering og optimering af rutedisponering.

Systemets understøttelse af to-vejs realtidskommunikation gør det muligt for koordinatorer at ændre ruter direkte fra webappen og kommunikere med chauffører uden forsinkelser. Det kan forbedre effektivitet, fleksibilitet og kvalitet i den daglige drift.

Derudover har systemet organisatoriske og samfundsmæssige perspektiver. Et mere præcist realtidsbillede kan reducere fejl, forbedre arbejdsgange og lette koordinatorers arbejdsbelastning. Effektiv ruteplanlægning kan samtidig mindske brændstofferbrug og driftsomkostninger og dermed bidrage til reduceret CO₂-udledning.

Samlet set viser projektet, hvordan realtidsdata og digitale værktøjer kan understøtte en mere effektiv, fleksibel og bæredygtig flådedrift og danner et solidt udgangspunkt for videre udvikling.

14 Konklusion

Projektets formål var at undersøge, hvordan man kan give flådekoordinatorer og chauffører et samlet, realtidsopdateret overblik over køretøjsstatus, ruter og beskeder, så de kan træffe hurtige beslutninger og reagere rettidigt på hændelser. Med udviklingen af Fleet Insight demonstrerer projektet, at dette er teknisk muligt gennem et integreret system bestående af webapp, Web API og simulator.

Løsningen opfylder problemformuleringen på flere centrale områder. Webappen giver et samlet og overskueligt overblik over flåden gennem et live-opdateret kort og detaljerede køretøjsdata. Realtidskommunikationen via SignalR sikrer, at både ruteændringer, telemetridata og alarmer vises øjeblikkeligt, hvilket understøtter hurtig beslutningstagning. Chatfunktionen muliggør direkte kommunikation mellem chauffører og koordinatorer, og alarmsystemet gør det muligt at reagere hurtigt på kritiske hændelser såsom lavt brændstof, høj hastighed eller offline-status. Simulatoren har desuden gjort det muligt at verificere hele realtidsflowet uden fysisk hardware.

Selvom projektets MVP opfylder problemformuleringen, peger arbejdet også på nogle naturlige begrænsninger ved den nuværende løsning. Systemets skalerbarhed er kun testet på lokal hardware, så det er endnu uklart, hvordan realtidskommunikationen vil fungere ved større flåder. Da telemetridata kommer fra en softwarebaseret simulator, kan systemets performance heller ikke fuldt ud vurderes i virkelige driftssituationer. Derudover vil funktioner som mere avanceret hændelseshåndtering og et egentligt analytics-dashboard kunne styrke systemets evne til at understøtte beslutningstagning yderligere.

Opdelingen af systemet i tre selvstændige dele har vist sig effektiv og har skabt en fleksibel arkitektur, der er velegnet til udvidelse. Den agile arbejdsproces har bidraget til en stabil fremdrift og en samlet, velfungerende MVP.

Samlet set viser projektet, at Fleet Insight kan skabe det nødvendige realtidsbillede af en flåde, som problemformuleringen efterspørger. Systemet fungerer som et solidt udgangspunkt for videreudvikling og kan med kommende forbedringer implementeres i praktiske driftsscenerier.

15 Fremtidigt arbejde

Dette afsnit præsenterer de mest centrale områder for videreudvikling af Fleet Insight. Punkterne tager udgangspunkt i de begrænsninger, der blev identificeret i diskussionen samt de muligheder, som systemets arkitektur åbner for.

15.1 Integration med fysisk hardware

Som beskrevet i afsnit [13.2 Styrker og begrænsninger](#) har projektet kun testet telemetri i et simuleret miljø. Et væsentligt næste skridt er derfor integration med reel GPS- og IoT-hardware for at klarlægge faktiske dataformater, opdateringsfrekvenser og kommunikationsprotokoller. Dette omfatter også vurdering af omkostninger ved hardware, installation og datatransmission.

Derudover blev systemets skalerbarhed kun testet på lokale personlige computere, hvor webappen viste ustabilitet ved høj belastning. Fremtidigt arbejde bør derfor inkludere performance- og stress-tests i miljøer med større ressourcer for at undersøge, hvordan Web API, database og SignalR håndterer større flåder. Det kan blive nødvendigt at indføre load balancing, opdele hubs eller optimere databaseforespørgsler for at sikre en produktionsklar drift.

15.2 Udvidet funktionalitet

Som fremhævet i afsnit [13.2 Styrker og begrænsninger](#) er både rutehåndtering og kommunikation centrale områder, der kan udvides. Rutehåndteringen understøtter i dag kun en enkelt destination. Fremtidigt arbejde kan derfor omfatte mellemstop, alternative ruter, prioriterede ruter, trafikbaseret optimering samt eventuel integration med kommercielle routing-tjenester for øget nøjagtighed.

Chatfunktionen kan ligeledes udvides med gruppechat, filoverførsel og mere avanceret håndtering af beskedhistorik. Alarmmodulet kan forbedres med flere alarmtyper, automatisk eskalering og bedre filtreringsmuligheder. Disse udvidelser vil styrke systemets anvendelighed i en praktisk driftskontekst.

15.3 Sikkerhed og databaseskyttelse

For at Fleet Insight kan anvendes i en produktionskontekst, skal systemet overholde gældende krav til datasikkerhed og GDPR. Dette indebærer implementering af adgangskontrol, logning, kryptering af data både i transit og i hvile samt løbende overvågning af sikkerhedshændelser.

Hertil bør der etableres organisatoriske retningslinjer for håndtering af brugerdata og rettighedsstyring.

15.4 Brugerinddragelse og pilottest

Et naturligt næste skridt er at afprøve Fleet Insight i samarbejde med en virksomhed eller organisation, der arbejder med flådedrift. En sådan pilottest vil give indsigt i virkelige brugerbehov, arbejdsgange og potentielle udfordringer, som ikke kan identificeres gennem simulerede tests. Feedback fra koordinatorer og chauffører vil være central for prioritering af videre udvikling og vurdering af systemets praktiske anvendelighed.

16 Appendiks

Bilag

1 Kravspecifikation

- 1.a User stories.pdf
- 1.b Ikke-funktionelle krav.pdf

2 Arkitektur

- 2.a Sekvensdiagrammer
 - 2.a.a Systemsekvensdiagrammer.pdf
 - 2.a.b Applikationssekvensdiagrammer.pdf
- 2.b C4 diagrammer
 - 2.b.a C4-diagrammer.pdf
- 2.c Andre diagrammer
 - 2.c.a Diverse diagrammer.pdf

3 Design

- 3.a Observer-pattern.pdf

4 Analyse

- 4.a Analyse arbejde.pdf
- 4.b Teknologianalyse.pdf

5 Implementering

- 5.a Web API source code
- 5.b Webapp source code
- 5.c Simulator source code
- 5.d Trelags-arkitektur.pdf
- 5.e Simulator i detaljer.pdf
- 5.f Alle webapp pages.pdf

5.g Detaljeret beskrivelse af implementering af login flow.pdf

5.h CI-pipeline.pdf

5.i CSV i detaljer.pdf

5.j how to run.pdf

6 Metode og Proces

6.a Tidsplan.png

6.b Metode og proces.pdf

6.c Backlog.pdf

6.d Arbejdstider for gruppen.pdf

7 Test

7.a Resultater af accepttest.pdf

7.b Smoke test af simulatoren.pdf

7.c Smoke test af Web API.pdf

8 Diverse

8.a brainstorm.jpg

9 Vejledningsmøder og referater

9.a Vejledningsmøder

9.a.a Vejledningsmøde 04-09-2025.pdf

9.a.b Vejledningsmøde 30-09-2025.pdf

9.a.c Vejledningsmøde 04-10-2025.pdf

9.a.d Vejledningsmøde 05-12-2025.pdf

9.b Referater

9.b.a referat 04-09-2025.pdf

9.b.b referat 30-09-2025.pdf

9.b.c referat 04-10-2025.pdf

9.b.d referat 05-12-2025.pdf

Referencer

- [1] YellowSoft. „The Benefits of Real-Time Tracking in Enhancing Fleet Management.“ (accessed: 10.10.2025). (2024), webadr.: <https://www.yellowsoft.com/blog/the-benefits-of-real-time-tracking-in-enhancing-fleet-management/>.
- [2] T. T. Services. „Real-Time Data, Real-World Impact: The Benefits of Telematics.“ (accessed: 10.10.2025). (2024), webadr.: <https://www.tip-group.com/en/knowledge/real-time-data-real-world-impact-benefits-telematics>.
- [3] Bringoz. „The Benefits of Real-Time Fleet Monitoring.“ (accessed: 09.10.2025). (2024), webadr.: [https://www.bringoz.com/the-benefits-of-real-time-fleet-monitoring/](https://www.bringoz.com/the-benefits-of-real-time-fleet-monitoring).
- [4] 3DTracking. „The Benefits of a Fleet Management System.“ (accessed: 11.10.2025). (2025), webadr.: [https://3dtracking.com/2025/11/11/benefits-of-fleet-management-system/](https://3dtracking.com/2025/11/11/benefits-of-fleet-management-system).
- [5] Canadian Trucking Alliance. „2016 Pre-Budget Submission to the House of Commons Standing Committee on Finance.“ (accessed: 10.12.2025). (2016), webadr.: <https://www.ourcommons.ca/Content/Committee/421/FINA/Brief/BR8118737/br-external/CanadianTruckingAlliance-e.pdf>.
- [6] *Python Documentation*, <https://docs.python.org/3/>, (accessed: 03.10.2025), Python Software Foundation.
- [7] *OSRM Documentation*, <http://project-osrm.org/docs/>, (accessed: 04.10.2025).
- [8] O. Standard, *MQTT Version 3.1.1*, <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>, (accessed: 02.10.2025), 2014.
- [9] *MQTT and WebSockets in IoT Architectures*, <https://iot.eclipse.org/resources/white-papers/mqtt-and-websockets/>, (accessed: 01.10.2025), Eclipse IoT Whitepaper.
- [10] Microsoft Corporation. „Introduction to ASP.NET Core.“ (accessed: 07.10.2025), Microsoft. (2024), webadr.: <https://learn.microsoft.com/aspnet/core/?view=aspnetcore-8.0>.
- [11] OpenJS Foundation. „Introduction to Node.js.“ (accessed: 06.10.2025), Node.js Foundation. (2024), webadr.: <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>.
- [12] Meta Platforms, Inc. „React Documentation.“ (accessed: 08.10.2025), React Team. (2024), webadr.: <https://react.dev/>.
- [13] Google LLC. „Angular Documentation.“ (accessed: 08.10.2025), Angular Team. (2024), webadr.: <https://angular.dev/>.

-
- [14] Geofabrik GmbH. „Geofabrik OpenStreetMap Data Downloads.“ (accessed: 15.10.2025). (2025), webadr.: <https://download.geofabrik.de/>.
 - [15] OSRM Project. „OSRM Backend — Docker Image.“ (accessed: 15.10.2025). (2025), webadr.: <https://hub.docker.com/r/osrm/osrm-backend/>.
 - [16] Eclipse Foundation. „Eclipse Mosquitto — Official Docker Image.“ (accessed: 15.10.2025). (2025), webadr.: https://hub.docker.com/_/eclipse-mosquitto.
 - [17] Project OSRM. „OSRM Backend.“ (accessed: 15.10.2025). (2025), webadr.: <https://github.com/Project-OSRM/osrm-backend>.
 - [18] Movable Type Scripts. „Latitude/Longitude Distance Calculation.“ (accessed: 15.10.2025). (2025), webadr.: <https://www.movable-type.co.uk/scripts/latlong.html>.