Markus Randa
Jan Anton Lande Pedersen
Lars Øyvind Ous

Release Note System

Bacheloroppgave i Dataingeniør
Veileder: Girts Strazdins
Mai 2020

Bacheloroppgave

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

NTNU
Kunnskap for en bedre verden

Markus Randa
Jan Anton Lande Pedersen
Lars Øyvind Ous

Release Note System

Bacheloroppgave i Dataingeniør
Veileder: Girts Strazdins
Mai 2020

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

NTNU
Kunnskap for en bedre verden

Release Note System

Markus Randa

Lars Øyvind Ous

Jan Anton Lande Pedersen

Mai 2020

Bachelor-oppgave

Department of ICT and Natural Sciences

Norwegian University of Science and Technology

Supervisor 1: Girts Strazdins

Supervisor 2: Anniken Karlsen

Obligatorisk egenerklæring/gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

Du/dere fyller ut erklæringen ved å klikke i ruten til høyre for den enkelte del 1-6:		
1.	**Jeg/vi erklærer herved at min/vår besvarelse er mitt/vårt eget arbeid, og at jeg/vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.**	☒
2.	**Jeg/vi erklærer videre at denne besvarelsen:** • ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • ikke refererer til andres arbeid uten at det er oppgitt. • ikke refererer til eget tidligere arbeid uten at det er oppgitt. • har alle referansene oppgitt i litteraturlisten. • ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	☒
3.	Jeg/vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§14 og 15.	☒
4.	Jeg/vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert i Ephorus, se **Retningslinjer for elektronisk innlevering og publisering av studiepoenggivende studentoppgaver**	☒
5.	Jeg/vi er kjent med at høgskolen vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens studieforskrift §31	☒
6.	**Jeg/vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider**	☒

Forord

Vi vil gjerne takke

- Girts Strazdins, som har gjort en knakende god jobb med å gi gruppen råd gjennom prosjektet, der det i hovedsak har gått på å hjelpe gruppen med rapportskrivning, kontrakter, sprint-møter.
- Cordel, som har gitt tydelig tilbakemelding gjennom prosjektet, slik at gruppen til en hver tid hadde god oversikt over tilstanden til produktet.
- Arne Styve, som hjulpet gruppen med å forme gode user stories og gitt innsikt i estimering av user stories.

Sammendrag

I en bransje der det er høyt fokus på korte leveranser av programvare, er det å dokumentere disse leveransene fortsatt tungvindt manuelt-arbeid. Oppgaven gitt av Cordel beskriver et ønske om et web-system som kan kan brukes til å effektivt dokumentere endringer i produkter, og å publisere disse endringene. Løsningen skal være både mer enkel og effektiv enn prosessen som brukes til dette i dag.

Vi har tatt i bruk moderne frontend og backend teknologier som inkluderer React, ASP.NET Core og Docker, med anerkjente design-prinsipp. Det leseren vil få innblikk i er hvordan denne prosessen er gjennomført, hvilke valg som er gjort og hvordan valgene ble implementert.

Løsningen som har blitt utviklet inkluderer et full-stack-system bestående av et REST API, med et frontend som implementerer verktøy for å løse den nevnte problemstillingen. Dette inkludererer blant annet tilpassede administrative verktøy, integrering med eksterne API-er, i en utvidbar arkitektur.

Summary

In an industry with a focus on short delivery of software, there is no time to waste when it comes to documenting these deliveries. The task given by Cordel describes a wish of a web-system that can be used to effectivly document all changes in product, and publish these changes. This solution is going to be more efficient and make it easier to use, compared to the solution Cordel has today.

We have made good use of modern front end and back end technologies like React, ASP.NET Core and Docker, with recognized design-principles. The reader will get an overview of how this process, where the choices made, and how they were implemented will be mentioned.

The solution that has been developed includes a full stack system consisting of REST API, with a front end that implements tools to solve this issue. This includes using customized tools, integration of remote API's, in a expandable architecture.

Innhold

Acknowledgement . . . i
Sammendrag . . . ii
Acronyms . . . 2

1 Introductions **5**
1.1 Rapportstruktur . . . 5
1.2 Introduksjon . . . 5
1.3 Bakgrunn . . . 6
1.4 Problemstilling . . . 6
1.5 Målsettning . . . 6
1.6 Avgrensninger . . . 6

2 Teoretisk bakgrunnskunnskap **8**
2.1 Pull Request . . . 8
2.2 Code Review . . . 8
2.3 Smidig Utviklingsmetodikk . . . 8
2.3.1 Scrum . . . 8
2.3.2 Scrum-board . . . 8
2.3.3 Scrum-roller . . . 9
2.3.4 Sprint-møter . . . 9
2.3.5 Story points . . . 10
2.3.6 Planning-poker . . . 10
2.3.7 User Story . . . 10
2.3.8 Backlog . . . 10
2.3.9 Velocity . . . 10
2.4 Design Mønstre . . . 10
2.4.1 Single-page application . . . 10
2.4.2 Trelags-arkitektur . . . 11
2.4.3 Single Source of Truth . . . 11
2.4.4 Seperation of Concerns . . . 11
2.4.5 Repository pattern . . . 11
2.4.6 Interface driven design . . . 12
2.5 Teknikker i ASP.NET Core . . . 12
2.5.1 ASP.NET Core Middleware . . . 12
2.5.2 ASP.NET Core Service . . . 12

2.5.3 Dependency injection 13
2.5.4 Contructor Injection 13
2.5.5 Unit of work 13
2.5.6 Automapper 13
2.6 Sikkerhet 14
2.6.1 JWT 14
2.6.2 Stateless Authentication 14
2.6.3 Data Authorization 15
2.7 REST 15
2.7.1 HTTP-metoder 15
2.7.2 HTTP-status-koder 15
2.7.3 Query-parameter 15
2.7.4 Path-parameter 15
2.8 Databaseteori 15
2.8.1 Nøkler 16
2.8.2 Primary key 16
2.8.3 Foreign key 16
2.8.4 Composite key 16
2.8.5 Forhold mellom tabeller 16
2.9 Reverse-proxy 17
2.10 Mikrotjenester 17
2.10.1 Container 17
2.10.2 Docker og Docker-Compose 17
2.11 Brukertesting 17
2.11.1 Unmoderated Remote Usability Testing 17
2.11.2 Lab usability testing 18
2.11.3 Guerilla Testing 18
2.12 Azure DevOps 18
2.12.1 Azure DevOps 18
2.12.2 Work Item 18
2.12.3 Azure Pipelines 18
2.13 Rammeverk og teknologier 19
2.13.1 TypeScript 19
2.13.2 Github-plugins 19
2.13.3 React-testing-library 19
2.13.4 Jest 19

2.13.5 Mocha og Chai . . . 19
2.13.6 ASP.NET Core . . . 20
2.13.7 Entity Framework Core . . . 20
2.13.8 Hibernate . . . 20
2.13.9 Postman . . . 20
2.13.10 React . . . 20
2.13.11 Rammeverk og biblioteker for React . . . 21
2.13.12 Memoization . . . 23
2.13.13 Callback . . . 23
2.13.14 Profiler . . . 23
2.14 Testing . . . 24
2.14.1 Unit Testing . . . 24
2.14.2 Conformance Testing . . . 24
2.14.3 Integration Testing . . . 24

3 Metode **25**
3.1 Programvare og Verktøy . . . 25
3.1.1 Postman . . . 25
3.2 Prosjektstyring . . . 26
3.2.1 Pull Requests . . . 26
3.2.2 Code Review . . . 26
3.2.3 Smidig Utviklingsmetodikk . . . 26
3.3 Planlegging . . . 28
3.3.1 Wireframes . . . 28
3.3.2 Gantt-diagram . . . 28
3.4 Prosjektarkitektur . . . 28
3.4.1 Presentasjons-laget . . . 29
3.4.2 Business og data-laget . . . 32
3.4.3 Mikrotjenester . . . 35
3.4.4 Reverse-proxy . . . 37
3.5 Funksjonalitet . . . 39
3.5.1 Teksteditor . . . 39
3.5.2 Drag-and-drop . . . 39
3.6 Sikkerhet . . . 39
3.6.1 Brukerautorisering . . . 39
3.6.2 Dataautorisering . . . 40
3.7 Testing . . . 41

3.7.1 Azure Pipelines . . . 41
3.7.2 Integration Testing . . . 41
3.7.3 Unit-testing . . . 41
3.7.4 Conformance Testing . . . 41
3.7.5 Brukertesting . . . 41

4 Resultat **44**

4.1 Funksjonalitet . . . 44
4.1.1 Redigering av Releases & Release Notes . . . 44
4.1.2 Offentlig side . . . 47
4.1.3 Administrator-side - Release Note System . . . 50
4.1.4 Administrator-side - Azure DevOps . . . 51
4.1.5 Login . . . 52
4.1.6 UI Tilbakemeldinger . . . 53
4.1.7 REST API . . . 54
4.2 Frontend-arkitektur . . . 54
4.2.1 React . . . 55
4.2.2 Komponentdesign . . . 56
4.2.3 Optimalisering av Ytelse . . . 57
4.2.4 Bruk Av Axios . . . 58
4.2.5 Tilstandsbehandling . . . 59
4.2.6 Testing . . . 63
4.2.7 Visning av releases . . . 64
4.3 Backend-arkitektur . . . 64
4.3.1 ASP.NET Core . . . 64
4.3.2 Test resultater . . . 70
4.3.3 Test har feilet . . . 70
4.3.4 Test er vellykket . . . 72
4.3.5 Oppbygging . . . 73
4.4 Database . . . 73
4.4.1 Konfigurasjon i ASP.NET Core . . . 73
4.4.2 Tabeller og Relasjoner . . . 75
4.4.3 Proxy . . . 78

5 Diskusjon **79**

5.1 Teknisk resultat . . . 79
5.1.1 Release Note Editor . . . 79

5.1.2 Release Editor . . . 79
5.1.3 Adminpanel . . . 80
5.1.4 Offentlige sider . . . 80
5.1.5 Frontend . . . 80
5.1.6 Backend . . . 82
5.1.7 Eksisterende Løsninger . . . 85
5.2 Gjennomførelese av prosjektet . . . 85
5.2.1 Smidig utvikling . . . 85
5.2.2 Versjonskontroll . . . 86

6 Konklusjon **88**
6.1 Resultatet . . . 88
6.1.1 Gjenstående arbeid . . . 89
6.2 Samarbeid . . . 89

Appendices **90**

Bibliografi **99**

Terminology

API Application Programming Interface, gjør at utvalgte funksjoner i et program kan kjøres av et annet program.

Continuous Integration Utviklere benytter seg av automatiserte tester for å forsikre seg at de endringer som de gjør, ikke ødelegger funksjonalitet og egenskaper på produktet.

Continuous Delivery Handler om å få den nyeste versjonen av produktet fort ut til kunden. Automatisering av lanserings prosessen slik at den kan skje ved få trykk.

Container Et miljø for å kjøre programvare som er isolert fra annen programvare.

Docker Et verktøy for sette og kjøre programmer i containere.

Wireframes Skisse som i grove trekk forklar stukturen til en nettside.

User Story En enhet arbeid innen smidig-utvikling som beskriver hva slags funksjonalitet som skal implementeres.

Business Logic Logikken som håndterer reglene for hvordan data kan skapes, leses, og endres.

GUI Graphical User Interface, grafisk brukergrensesnitt

UI User interface, brukergrensesnitt

Assertion En forutsetning gjort i sammenheng med testing.

Routing Routing er prosessen å matche en URL til en software komponent. Routing er en essensiell del av SPA applikasjoner.

Work Item Her er det snakk om en Work Item fra Azure DevOps, som definerer en enhet arbeid i sammenheng med Agile-Metoder.[34]

Request En HTTP-Request.

Response En HTTP-Response.

Boilerplate Et uttrykk som benyttes til å beskrive lik kode som må plasseres rundt om i programmet for at det skal fungere.

Statically Typed Beskriver en kodestil som blant annet krever at alle datatyper blir deklarert i compile-time.

Abbreviations

CI/CD Et samlebegrep for continous integration, continous delivery og deployment

MVP Minimal Value Product. Minstekravet for prosjektet

REST REpresentational State Transfer

URI Uniform Resource Identifier

HTTP HyperText Transfer Protocol

SQL Structured Query Language

ER Entity-Relation

GDPR General Data Protection Reguluation

UX User Experience, opplevelsen av brukergrensesnittet for en bruker

WYSIWYG What you see is what you get. Word-lignende tekst editorer.

ORM Object-Relational Mapping

Figurer

2.1 Eksempel på et *Repository Pattern.* 12
2.2 Eksempel på et *Unit of work pattern.* 13
2.3 Sekvensdiagram som viser flyten i en vellykket autentisering. 14
2.4 Flytdiagram som viser hvordan UI og Redux henger sammen. 22

3.1 Eksempel på bruk av Azure DevOps scrum-board. 28
3.2 Oversikt over mikrotjenester i test-miljø. 36
3.3 Oversikt over mikrotjenester i prod-miljø. 37
3.4 Oversikt over mikrotjenester i utviklings-miljø. 38
3.5 Flyt-diagram som viser eksempel på uthenting av Release Note med data-autorisering. 40
3.6 Tilbakemeldingsskjema 42

4.1 Oversikt over sammenhengen mellom Release og Release Note 44
4.2 Release Note Editor 45
4.3 Release Editor 46
4.4 Utseende på forsiden til nettsiden. 47
4.5 Oversikt over alle releases tilgjengelig for valgt produkt. 48
4.6 Release artikkel 49
4.7 Admin-side 50
4.8 Azure tab på admin side 51
4.9 Sammenheng mellom Release Note og Work Item 52
4.10 Eksempel på tilbakemelding via notifikasjon 53
4.11 Passord tilbakemelding 54
4.12 Oversikt over komponent-arkitektur 55
4.13 Routing-konfigurasjon 57
4.14 Graf som viser den globale tilstanden. 60
4.15 Informasjonsflyt med redux. 60
4.16 Test-rapport på test har feilet. 71
4.17 Test-rapport på test er vellykket. 72
4.18 ER-Diagram for Release Notes 76
4.19 ER-Diagram over Mapping 77
4.20 ER-Diagram over Auth 78

5.1 Oversikt som viser popularitet for Java EE og .NET Core på google over de fem siste årene. 83

Kapittel 1

Introduksjon

Dette kapittelet vil introdusere hva som har blitt jobbet med i prosjektet. Her blir det opplyst om hva som var årsaken til at oppgaven skulle løses, hva løsningen skal gjøre for oppdragsgiver og hvilke avgrensninger oppgaven har.

1.1 Rapportstruktur

Kapittel 1 - Introduksjon
Innledning til prosjektet med bakgrunninformasjon, problemstilling og avgrensninger.

Kapittel 2 - Teoretisk bakgrunnskunnskap:
Gir leseren tilstrekkelig med bakgrunnskunnskap for å kunne forstå resten av rapporten.

Kapittel 3 - Metode:
Inneholder en beskrivelse av metodene som er brukt i prosjektet for å løse oppgaven.

Kapittel 4 - Resultat:
Inneholder alle oppnådde resultat i prosjektet.

Kapittel 5 - Diskusjon:
En diskusjon av resultatet og metodene som ble brukt for å løse oppgaven.

Kapittel 6 - Konklusjon:
En sammenfatning av temaene diskutert i resultat og diskusjon - trekker en endelig konklusjon over utført arbeid.

1.2 Introduksjon

Cordel Norge AS er en bedrift som tilbyr IT-tjenester for mange norske bedrifter innen VVS, elektro og bygg. Cordel er akkurat nå i en overgangsfase for å bli mer sky-basert, samt implementere continuous integration og continuous deployment/delivery (CI/CD). Dette er en naturlig utvikling for å holde seg konkurransedyktig og moderne. En

naturlig følge av dette er en økning i antall oppdatering på tjenestene de tilbyr.

1.3 Bakgrunn

I nyere tid så blir sky-tjenester mer og mer vanlig. Det er også blitt populært med CI/CD. Hyppigheten i oppdatering og utgivelse av programvare har derav økt de siste årene, mens prosessen å dokumentere oppdateringerene henger etter.

1.4 Problemstilling

I en bransje der det er høyt fokus på korte leveranser av programvare, er det å dokumentere disse leveransene fortsatt tungvindt manuelt-arbeid. Det skal utvikles et moderne web-system som kan kan brukes til å effektivt dokumentere og publisere releasess. Løsningen skal være både mer enkel og effektiv enn prosessen som brukes til dette i dag.

1.5 Målsettning

Mål for prosjektet er

1. Designe og utvikle en moderne web-applikasjon. Web-applikasjonen skal være et verktøy som kan kan brukes til å effektivt dokumentere og publisere releases. Dette inkluderer å lage administrerende verktøy som forenkler denne arbeidsflyten.
2. Designe og utvikle et REST-API med database. Rest-APIet skal bestå av robuste metoder for å hente ut og behandle data i databasen. Databasen skal bestå av oversiktlige og fornuftige tabeller.
3. Lage et system som kan integreres mot Azure DevOps.
4. Implementere et system med anerkjente arkitektur-prinspipp og kode-praksis.

1.6 Avgrensninger

Cordel satte noen krav for hvilke teknologier som skulle benyttes i systemet. Disse kravene ble satt slik at Cordel har muligheten til å videreutvikle prosjektet.

Kravene var som følger:

1. Trelags arkitektur, backend, frontend og database
2. Backend skal følge REST prinsipper og være skrevet i C#
3. Databasen skal bruke PostgreSQL

4. Frontend skal helst bruke React

5. Prosjektet skal bruke Docker containere.

Siden dette skal integreres inn i Cordel sin arbeidsflyt må det fungere med verktøy de allerede har i bruk. Cordel bruker Azure DevOps for å holde styr på arbeidsoppgaver. Det betyr at systemet må kobles opp mot Azure DevOps.

Kapittel 2

Teoretisk bakgrunnskunnskap

I dette kapittelet skal leseren få muligheten til å sette seg inn nødvendig bakgrunnskunnskap for å kunne være i stand til å forstå det som kommer senere i rapporten.

2.1 Pull Request

Når en utvikler har endringer eller tillegg som den ønsker skal være en del av kildekoden, må det bli gjennomført via en pull request. En pull request er en forespørsel om å legge til endringer i kildekoden. Den inneholder alle commits som utvikleren ønsker å legge til i kildekoden. Pull requests kan også inneholde en en beskjed som utdyper hvorfor eller hva som skjer. [1]

2.2 Code Review

En Code Review er en gjennomgang av en Pull Request, der tilbakemeldinger og diskusjoner tar sted. Når en pull request blir lagt inn, så leser andre utvikerere gjennom koden og gir tilbakemeldinger.

2.3 Smidig Utviklingsmetodikk

I dette underkapittelet skal begreper og metoder som er relevante for å løse oppgaven beskrives, der grunnprinsipper som Scrum og teknikker som underbygger scrum blir forklart.

2.3.1 Scrum

Scrum er en smidig arbeidsmetodikk der arbeid defineres gjennom User Stories, gjennomføres i iterasjoner, har noen definerte roller og har fokus på kontinuerlig forbedring. Alle arbeidsoppgavene blir definert som user stories, som igjen blir lagt til i en product backlog. I hver iterasjon eller sprint, som det kalles i Scrum er det definert fire forskjellige møter. Det finnes tre roller i Scrum. Det legges stor vekt på at man skal ha kontinuerlig forbedring gjennom hele utviklingsprosessen, der kunden er i fokus og blir inkludert i hele prosessen. [69]

2.3.2 Scrum-board

Et Scrum-board er et visuelt hjelpemiddel, ofte i form av et tabulært skjema, for å holde oversikt over hvilke oppgaver og hvilken tilstand de befinner seg i. Skjema inneholder user stories, som deles opp i mindre tasks som en

utvikler utfører. Disse taskene kan befinne seg i ulike tilstander som todo, in-progress, ready for review og done.

2.3.3 Scrum-roller

I Scrum er det definert tre forskjellige roller som består av en Scrum-master, en produkteier og et utvikligsteam.

Scrum-master har som oppgave å sørge for at prinsippene definert for Scrum blir implementert på skikkelig vis, der jobben består av å hjelpe både utviklingsteamet og produkteier med å bruke Scrum for å oppnå forretningsverdi.

Produkteier er den som eier product-backloggen og har ansvar for hvilke oppgaver som blir lagt til. Produkteier skal kontinuerlig sørge for at product-backloggen er prioritert rett. Et annet viktig poeng er at det er produkteier som står med profitt- og tapsrisiko i prosjektet, som forutsetter at det utvikles et kommersielt produkt.

Utviklingsteamet er en gruppe på 3 til 9 personer som står for å utvikle produktet. Dette teamet kan forskjellige faggrupper, der det ikke bare er utviklere men også mulighet for designere til å delta. Utviklingsteamet skal være selvstyrt som betyr at det ikke defineres en spesifikk leder for utviklgsteamet. [9]

2.3.4 Sprint-møter

Det er definert fire forskjellige sprint-møter der sprint-review, sprint-planning og retrospective gjennomføres en gang hver sprint, og stand-up hver dag.[71]

Sprint-planning

Er det første møtet i en sprint der utviklingsteamet og produkteier blir enige om prioriteringen på user stories, og bestemmer hva som skal bli gjort basert på hvor mange story points utviklingsteamet er i stand til å utføre.

Sprint Review

Et møte som tar sted ved sprintslutt, der utviklingsteamet gjennomgår hva som har blitt gjort i løpet av en sprint sammen med produkteier. Her blir det også en anledning for utviklingsteamet å forklare hvorfor sprinten har gått som den har gått, og produkteier kan gi tilbakemelding på hva som tenkes rundt produktet så langt.

Retrospective

Et møte som tar sted i sprintslutt der bare utviklingsteamet drøfter hva som kunne ha vært gjort annerledes, hva som gikk bra også til slutt definere konkrete tiltak til forbedring.

Stand-up

Et daglig møte der hver utvikler forteller hva som har blitt gjort dagen før med mulighet til å ta opp hindringer som har oppstått, etter det forteller utvikler hva som skal fortsettes med resten av dagen.

2.3.5 Story points

En abstrakt verdi som representerer innsatsen som kreves for å fullføre en user story. Denne verdien representerer innsatsen som kreves av utviklingsteamet som setter verdien, det vil si at den ikke kan brukes for å sammenligne forskjellige teams.[14]

2.3.6 Planning-poker

Dette er en estimeringsteknikk som benyttes for estimere story points til en User Story. I korte trekk så benytter denne teknikken kort med tall fra 0 til 100 som representerer story points, der hver deltaker viser et frem et kort som representerer innsatsen som kreves for å løse diskutert user story. [15]

2.3.7 User Story

En user story er en enkelt arbeidsoppgave som holder på informasjon om hva som skal gjøres, hvor omfattende jobben er og hvilke akeptansekrav den har. Et eksempel på hvordan beskrivelsen av arbeidsoppgaven kan være er Som en bruker får jeg tilbakemelding ved feil passord på forsøkt innlogging". Det man kan legge merke til i eksempelet er at man alltid skriver opp hvem som gjør en handling ettefulgt av hva som skal skje når vedkommende gjør en valgt handling. Story points benyttes for å forklare hvor mye innsats som skal til for å løse oppgaven. Akseptansekravene er til for å validere at user storien er gjennomført. [16]

2.3.8 Backlog

En backlog er en samling av oppgaver som skal gjøres. Det finnes i hovedsak to typer backlogs i Scrum; Product-backlog som er en samling av oppgaver for hele produktet som utvikles, og Sprint-backlog som er samlingen som bare gjelder en spesifikk sprint. [72]

2.3.9 Velocity

Velocity er et måleenhet innen smidig-utvikling som enten er basert på tid eller innsats som kreves for å løse et problem. På dette viset får man en konstant enhet man kan bruke til å sammenligne og planlegge.[75]

2.4 Design Mønstre

I dette underkapittelet blir det gjennomgått design mønstre som er relevant for denne oppgaven.

2.4.1 Single-page application

En single-page applikasjon er en webside som aktivt oppdaterer innholdet på siden med ny data fra en web server. Dette er ulikt standard metoden hvor man laster inn hele sider omgangen. I single-page applikasjoner kan mindre

deler av siden oppdateres slik at websiden føles mer som en native applikasjon. Single-page applikasjoner benytter routing for å oppnå flytt i applikasjonen. Diverse handlinger fører til endringer i URL-en, endring i URL fører til at ulike programvare komponenter blir aktive. På dette viset oppnår single-page applikasjoner høy ytelse til tross for mengden dynamisk innhold. [48]

2.4.2 Trelags-arkitektur

En lag-basert arkitektur i software utvikling som deler opp biter av programvaren i separerte deler, der hver enkelt del kun trenger å forholde seg til seg selv. De forskjellige lagene behøver derfor ikke å bry seg om hvor de får data fra, men kan fokusere på selve dataen. Lagene i dette designet er stablet vertikalt, altså at hvert lag tilbyr en tjeneste til laget over seg. I en trelags-arkitektur er lagene definert som *Presentasjon*, *Business*, og *Data*.

Presentasjonslaget har som ansvar å vise det grafiske brukergrensesnittet, som presenterer data og kommuniserer med de andre lagene. Dette blir altså det øverste laget, og er direkte tilgjengelig for sluttbrukeren.

Business-laget er det midterste laget, og er ansvarlig for å behandle innkommende og utgående data. Dette inkluderer for eksempel å kombinere data-kilder, regne ut verdier eller styre tilgang.

Det nederste laget er Data-laget. Her er det endelige stedet hvor data lagres i en database.

Hensikten med dette design-mønsteret, er å forenkle utvikling og testing, ved å dele programvare opp i flere deler for å redusere omfanget og kompleksiteten av det som skal utvikles. [66]

2.4.3 Single Source of Truth

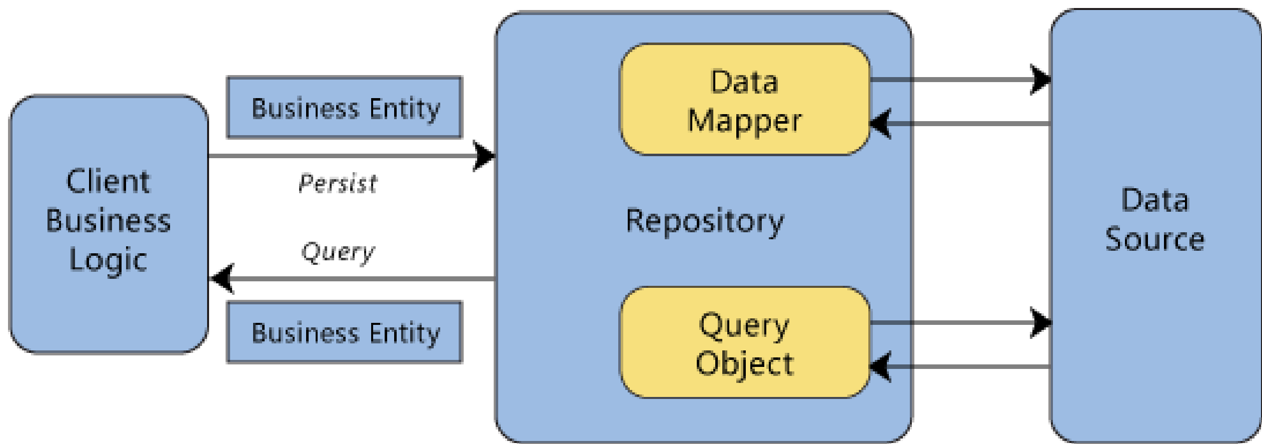
Data skal holdes på kun et sted. Data kan bli brukt og innhentet fra ulike moduler, men behandles kun på ett sted. Fordelene med dette er at man gjør det enklere å opprettholde korrekt data, og korrekt mengde data.[54]

2.4.4 Seperation of Concerns

Et design prinsipp som sier at kode skal bli separert etter hvilken hensikt de har. For eksempel så er det ikke gunstig å få problemmer med innlogging når en legger til ny funksjonalitet til release editoren. Det skal være et klart skille mellom kode for modul A, og kode for modul B.[17]

2.4.5 Repository pattern

Et mønster for å isolere data laget fra business logikk. Man definerer klasser som har ansvaret for innhenting og lagring av data for gitt entitet. Dette er ønskelig for flere grunner. Man minimerer risiko når en henter og behandler data, og man gjør det lettere å få lagd tester. Duplisering av kode blir redusert, og behandling av data blir sentralisert.[29]



Figur 2.1: Eksempel på et *Repository Pattern*.

[73]

2.4.6 Interface driven design

Er et mønster som innebærer å definere kontrakter for alle oppgaver som skal løses i et system, og deretter utvikle løsninger som oppfyller kontraktene. Det man oppnår med denne teknikken er kode som er løsere koblet der kommunikasjonen går gjennom interfaces, istedenfor direkte mellom klasser. [6]

2.5 Teknikker i ASP.NET Core

Her blir det gjennomgått et par teknikker og begreper som er relevante når man utvikler applikasjoner med rammerverket ASP.NET Core.

2.5.1 ASP.NET Core Middleware

Middleware i en ASP.NET Core applikasjon er en software-modul som tar hånd om alle HTTTP-requester som blir sendt til en applikasjonsserver før den blir videresendt nedover applikasjonen. Det denne middlewaren gjøre er å behandle alle HTTP-requester med en rekke funksjoner, de mest essensielle funksjonene for denne oppgaven er routing, autentisering og autorisering. Routing gjør at alle requester havner hos rett endepunkt, autentisering og autorisering sørger for at adgangen til ressurser blir regulert. [5]

2.5.2 ASP.NET Core Service

En slik service er en måte å definere en depency på i .NET Core. Måten dette blir gjort på er vanligvis gjennom å legge til alle services i oppstartsklassen til en ASP.NET Core-applikasjon. Her vil en service alltid være enten et interface eller en klasse. [78]

2.5.3 Dependency injection

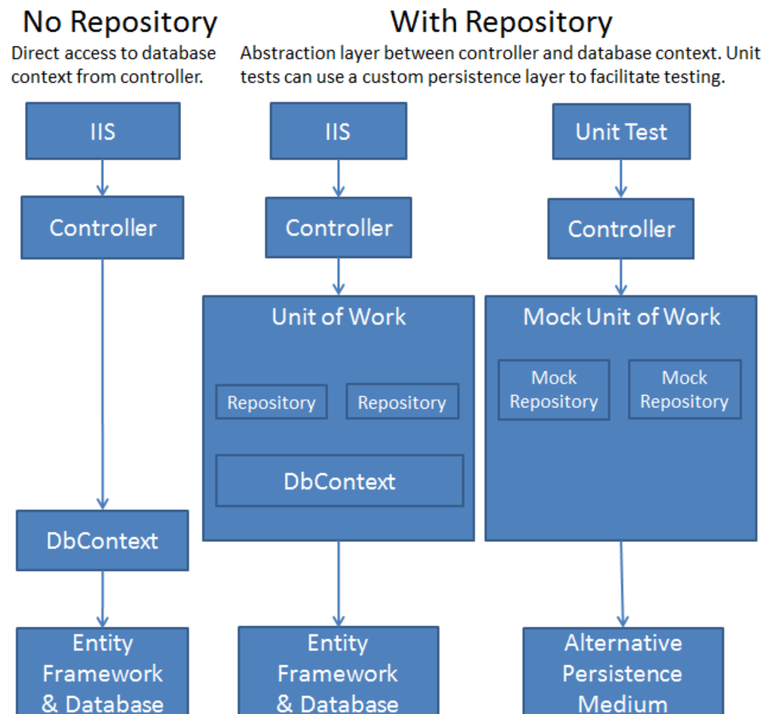
Dependency injection er et mønster hvor istedet for å la klassen som krever en dependency kalle på dependencyen, så vil dependency heller kalle klassen. Dette kombinert med bruk av interfaces gir løs kobling. Dette gjør det trivielt å sette inn mocks som dependency for å enklere få testet applikasjonen.[67]

2.5.4 Contructor Injection

Constructor injection er en teknikk i .NET Core som tillater utvikler å referer til en service som blir instansiert i konstruktøren til en klasse. [7]

2.5.5 Unit of work

Et mønster som gjør databehandling tryggere. Dette gjøres ved a la kun en operasjon skje om gangen. Dette gjøres via en in-memory liste som holder på transaksjonene som skal utføres. [73]



Figur 2.2: Eksempel på et *Unit of work pattern.*

[73]

2.5.6 Automapper

AutoMapper er et bibliotek som gjør jobben enklere med å mappe felt fra en klasse til en annen i C#-prosjekter. [10]

2.6 Sikkerhet

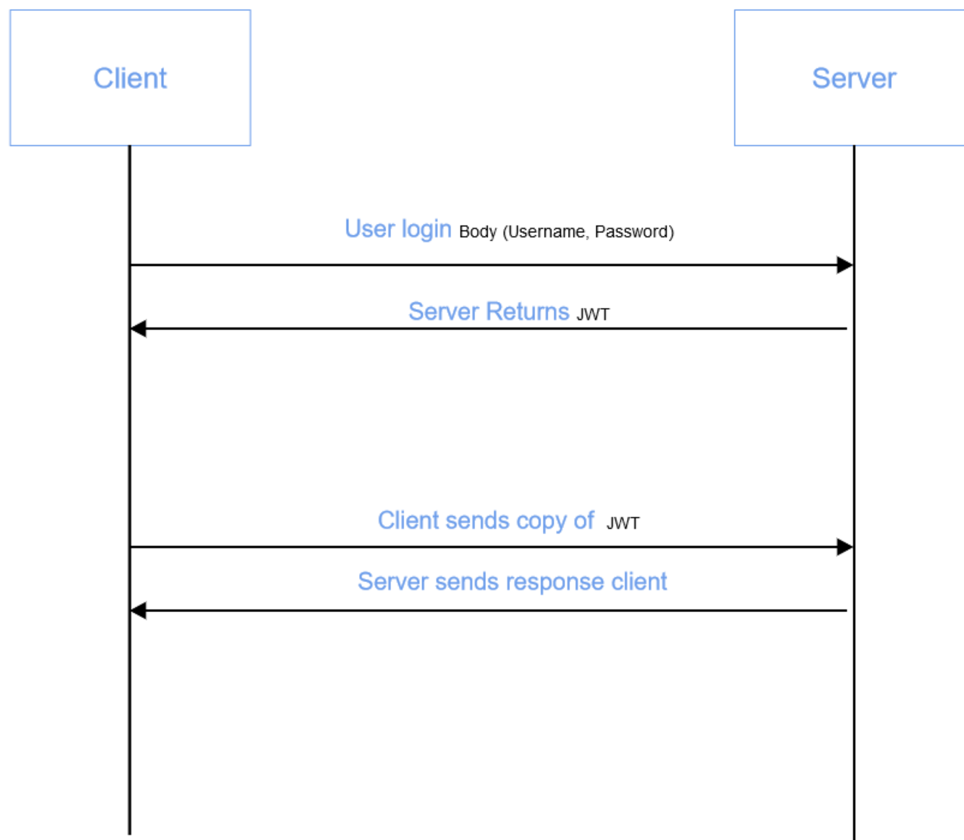
Det som skal diskureres er ymse teknologier eller metoder som benyttes for å gjennomføre både autentisering og autorisering.

2.6.1 JWT

JWT eller JSON Web Token er et alternativ til autentisering med passord og brukernavn. I praksis er måten denne brukes på at en autentisering-server utsteder en verifisert token med informasjon om bruker. Brukeren kan deretter oppgi denne når det oppstår behov for autentisering. [46]

2.6.2 Stateless Authentication

Stateless Authentication er en måte å oppnå autentisering uten at en bruker av et system til en hver tid må oppgi passordet sitt. Dette fungerer som et alternativ til å opprette et forhold mellom klient/tjener der man ikke oppretter en sesjon, men heller automatisk sender en token som forteller hvem brukeren er på en sikker måte. [47]



Figur 2.3: Sekvensdiagram som viser flyten i en vellykket autentisering.

2.6.3 Data Authorization

En måte å autorisere adgang til data der man autoriserer mot hver enkel entitet, i motsetning til for eksempel å autorisere på adgang til endepunkt.[70]

2.7 REST

REST er en arkitektur-stil som definerer hvordan klient-server kommunikasjon foregår. En klient vil sende en ressurs, ofte i form av en URI (Uniform Resource Identifier). For eksempel "api/bruker/id". Klienten vil også legge til en HTTP-metode. Serveren mottar forespørselen, behandler data og sender tilbake en response.[64]

2.7.1 HTTP-metoder

En metode som må inkluderes i en HTTP-request der metoden beskriver hva slags handling requesten forespør. De metodene som er tilgjengelige å bruke er GET, POST, PUT og DELETE, der hver av disse metodene representerer en handling. [76]

2.7.2 HTTP-status-koder

En standardisert kode som benyttes til å definere hva slags type response det er snakk om. En type response kan være 200, som betyr at alt er vellykket, og en annen kan være 401 som betyr at den som forespør mangler autorisering til å få adgang. [45]

2.7.3 Query-parameter

En del av URL-en til en HTTP-request som brukes for å inkludere mer utfyllende informasjon om requesten. Dette parameteret består av en nøkkel og en verdi, som for eksempel "units=metric". Dette paramteret må plasseres etter et spørsmålstegn i URL-en. [55]

2.7.4 Path-parameter

En del av URL-n til en HTTP-request som brukes for å inkludere mer informasjon om hva som forespørres. Dette parametere plassers sammen med adressedelen av en URL. Et eksempel kan være at man har et endepunkt Product, der man ønsker å referere til et produkt med id=1. Da kan man bygge opp API-en til å ta verdien fra URL slik: "api/products/{product_id}/". [52]

2.8 Databaseteori

Det som skal forklares i dette underkapittelet er forskjellige begreper og teknikker som brukes når man skal opprette og normalisere database.

2.8.1 Nøkler

I dette underkapittelet blir det gjennomgått hva de forskjellige nøklene brukt i relasjonsdatabaser er for noe.

2.8.2 Primary key

En Primary key er den kolonnen i en rad som identifiserer raden. Som regel er dette en numerisk verdi som ofte er generert av databasen.

2.8.3 Foreign key

En Foreign key er en kolonne i en rad som peker på en primary key i en annen tabell. Denne blir brukt som en referanse til en raden i en annen tabell.

2.8.4 Composite key

En Composite key er en samling av kolonner som brukes som primary key i en rad. Dette gjøres ofte i en sammenhenger der det er kombinasjonen av flere kolonner som naturlig identifiserer en rad.

2.8.5 Forhold mellom tabeller

Det som skal forklares i dette underkapittelet er forskjellige teknikker for å håndtere forhold i en database. Denne gjennomgangen inkluderer one-to-one, one-to-many og many-to-many.

One-to-one

Dette forholdet beskriver et forhold der man bruker en kolonne i en tabell til å peke på en rad i en annen tabell. Et eksempel på at dette kan være nyttig er hvis man har en tabell Urbefolkninger og en tabell Lokasjoner, så vil det være mulig for en urbefolkningsstamme å ha sin lokasjon med land og sted i en annen tabell, der lokasjonen ikke nødvendigvis blir slettet hvis urbefolkningsstammen blir fjernet fra tabellen. [50]

One-to-many

Dette forholdet beskriver et forhold der foreign key til en kolonne blir brukt en eller flere ganger i en annen tabell. Hvis man fortsetter eksempelet fra forrige avsnitt og sier at man ønsker en oversikt over alle verktøyene en urbefolkningsstamme bruker, så kan man bruke denne teknikken ved å legge til flere foreign keys i en ny tabell Verktøy, der hver rad representerer en urbefolkningsstamme og et verktøy. [49]

Many-to-many

Dette forholdet beskriver et forhold der flere rader i en tabell har et forhold til flere rader i en annen tabell. Her kan utvide eksempelet som gjennomgås i dette kapittelet med at flere urbefolkningsstammer bruker samme verktøy,

dette går an med forrige eksempel men da risikerer man data-duplisering. Så en løsning på dette vil være å opprette to nye tabeller Verktøy og Urbefolkningsverktøy, der både verktøy og urbefolkingsstammer har flere foreign keys i urbefolkingsverktøy. [37]

2.9 Reverse-proxy

En reverse-proxy er en tjener som videresender forespørsler fra utenfor det interne nettverket til interne tjenester. Reverse-proxyen vil da også fungere som gateway for disse interne tjenstene, slik at alle responser går gjennom den. En fordel med dette er at det øker sikkerheten ved å ikke eksponerer alle de interne tjeneste mot det eksterne nettverket.[65].

2.10 Mikrotjenester

Mikrotjenester er en arkitektur-stil som strukturer en applikasjon eller prosjekt til flere selvestendige komponenter. Denne arkitekturstilen er svært populær siden man oppnår løs kobling mellom andre komponenter, god skalering, vedlikeholdbar og testbar kode.[42]

2.10.1 Container

En container er en isolert prosess som kan kjøre et eget operativsystem definert av et image. Dette imaget er som regel laget for å ta opp minst mulig plass og inneholder kun software for å tilby akkurat en tjeneste. Et eksempel på en container kan være en instans av et image med PostgreSQL, der oppgaven til containeren blir å lagre på data og håndtere forespørsler. [24]

2.10.2 Docker og Docker-Compose

Docker er en plattform for å lage og håndtere containere. Docker-Compose er et tilleggsverktøy til Docker, som tilbyr tilleggsfunksjoner til styring og manipulering av containere.[22][23]

2.11 Brukertesting

Dette underkapittelet inneholder en oversikt over flere kjente teknikker innenfor brukertesting.

2.11.1 Unmoderated Remote Usability Testing

En uovervåket testeteknikk der brukere får fjerntilgang til produktet som skal testes, der utvikler får tilbakemelding fra test-subjekt etter testen er fullført. [44]

2.11.2 Lab usability testing

Denne metoden er en kvalitativ-metode hvor deltakere blir stilt spørsmål eller tildelt oppgaver, samtidig som at man lagrer data om erfaringen til deltakerne. Testingen foregår som regel i et avlukket lokale der deltakerne kan fokusere på det som skal testes. Årsaken til å velge denne test-metoden er hvis man er interessert i å få et godt innblikk i hvordan det som skal testes oppleves for brukeren, med mulighet til å stille spørsmål til brukeren fortløpende. [36]

2.11.3 Guerilla Testing

En testeknikk hvor en presenterer en prototype-versjon av systemet for tilfeldige personer, og får tilbakemelding på stedet. Det man er ute etter her er å se om applikasjonen er intuitiv å bruke og hurtig tilbakemleding fra mange subjekter er i fokus. Fordelen med slik testing er at man får testet et bredt spektrum av brukere ganske kjapt.[30]

2.12 Azure DevOps

I denne seksjonen skal det oppsummeres alle konsepter innenfor Azure DevOps som er relevant for denne oppgaven.

2.12.1 Azure DevOps

Azure DevOps er et samarbeidsverktøy fra Microsoft som inkluderer en rekke tjenester for å drive bl.a smidig-utvikling, versjonskontroll og CI/CD [12].

2.12.2 Work Item

En work item er en Azure DevOps-entitet som beskriver en bug, task eller user story. En work item inneholder informasjon knyttet til hva oppgaven er, hvem som arbeider med den, status (in progress, ready for review, done) og eventuelt dato for fullførelse.

2.12.3 Azure Pipelines

Azure Pipelines er en tjeneste i Azure DevOps for å bygge, teste og distribuere kode-prosjekter.

Pipeline Task

En task i Azure Pipelines er en innebygd implementasjon av en større oppgave som kan utføres i azure pipelines.

Exit Code

En metode som benyttes når et script returnerer fra eksekvering. Dette er ikke standardisert, men en vanlig konvensjon er å returnere 0 ved vellykket utførelse, og alle verdier over beskriver at noe har skjedd.[51]

2.13 Rammeverk og teknologier

Det som skal oppsummeres i dette underkapittelet er ymse rammeverk og teknologier som er relevant for denne oppgaven.

2.13.1 TypeScript

TypeScript er et superset av JavaScript. TypeScript innfører flere objekt-orienterte prinsipper og setter strengere krav enn JavaScript. TypeScript benytter disse kravene til å finne innføre type-safety, som gjør det mulig å oppdage feil ved kompilasjon.[74]

2.13.2 Github-plugins

Dette underkapittelet tar for seg alle Github-plugins som er relevante for oppgaven. Github-plugins er tilleggsprogram man kan bruke i Github-prosjekter.

DeepScan

DeepScan er en plugin i GitHub som scanner utviklings-prosjekter for vanlige runtime -og kvalitetsfeil [21]. De kvalitetsfeilene DeepScan sjekker er en liste over anerkjente konvensjoner som matches mot koden, her er det fullt mulig å velge hvilke konvensjoner som skal benyttes.

Mergify

Mergify er en plugin til GitHub som lar deg automatisere vanlige oppgaver i en pull request. Dette inkluderer blant annet merging, lukking av branch, kommentering, tildeling av reviewers, og mer.[41]

2.13.3 React-testing-library

Et bibliotek som gir mulighet til testing mot komponenter i React.[58]

2.13.4 Jest

Et komplett JavaScript test-rammeverk utviklet av Facebook, som kan brukes til å mocke komponenter og gjøre assertions.[33]

2.13.5 Mocha og Chai

Mocha er et fleksibelt test-rammeverk for JavaScript. Mocha er fleksibelt med tanke på at det er lagd for å ta i bruk tredjeparts assertion og mocking biblioteker. Chai er et slikt assertion bibliotek.[43] [13]

2.13.6 ASP.NET Core

ASP.NET Core er en implementasjon av .NET rammeverket med cross platform-støtte. Det man bruker dette rammerverket til hovedsaklig er å utvikle REST-api- og MVC-webapplikasjoner. [77]

2.13.7 Entity Framework Core

Entity Framework Core eller EF-Core er en implementasjon av Entity Framework, som hovedsaklig gjør ORM-mapping og databasebehandling. Rammeverket gjør jobben til utvikleren enklere ved å opprette database-tabeller og relasjoner basert på en objekt-orientert struktur. En annen ting rammeverket gjør er å gjøre det enklere å hente ut og manipulere dataen som ligger i databasen.[68]

Naming Conventions

Denne teknikken gir mulighet til å redusere boilerplate-kode i et par tilfeller. En ting som skjer automatisk når man setter opp klasser til å brukes i databasen, er at EF-Core bruker navnet på klassen i flertallsform som tabellnavn. En annen funksjon er at hvis man referer til en annen klasse som har blitt implementert i databasen, så bruker EF-Core dette til å opprette relasjoner. [25]

2.13.8 Hibernate

Et rammeverk som hovedsaklig gjør ORM-mapping og databasebehandling i Java-applikasjoner. Rammeverket tilbyr metoder for å opprette tabeller og forhold av definerte klasser. Databasebehandling får også et nytt grensesnitt gjennom API-er som Hiberante tilbyr, som igjen reduserer boilerplate-kode. [31]

2.13.9 Postman

Postman er et program som i hovedsak gjør det mulig å holde på en samling requests, men har flere avanserte støttefunksjoner. En kraftig funksjon i Postman er muligheten til å definere query-parameter, path-parameter og miljøvariabler. En annen viktig funksjon som finnes er muligheten til å dele på denne samlingen med requests i sanntid, der man kan definere teams med forksjellige roller. Den siste nevneverdige funksjonen er muligheten til å plukke ut relevant informasjon fra samling av requests og genere en api-dokumentasjon. [53]

2.13.10 React

React er et JavaScript-bibliotek for å lage brukergrensesnitt. React benytter virtuelt DOM for å lage egne React elementer. Brukegrensesnitt i react er bygd opp av et hierarki av komponenter. Komponenter kan ha egen tilstand, og kan interaktere med sin egen livssyklus. Komponenter kan også motta data i form av et objekt som kalles props. Alt i alt kan komponenter settes sammen og manipuleres for å skape dynamiske og interaktive brukergrensesnitt.[56]

Funksjonelle og klasse-baserte komponenter

React tilbyr to forskjellige måter å skrive komponenter på. Enten kan komponentene skrives som klasser, eller de kan skrives som funksjoner [28]. Disse to to måtene å implementere komponenter på, har også forskjellig API til React rammeverket.

Hooks

Hooks er måten en kommuniserer med React funksjoner fra funksjonelle komponenter. [79].

Dataflyt i React

React har en en-veis dataflyt [57], som vil si at data fra et komponent kun kan sendes nedover til underkomponenter.

2.13.11 Rammeverk og biblioteker for React

Dette underkapittelet tar for seg alle bibliotek og rammerverk som er relevante for oppgaven.

Create-react-app

CLI verktøy for å sette opp en single-page-application. Dette verktøyet tilbyr en enkel måte å sette opp denne applikasjonen uten behov for konfigurasjon.[20]

DraftJS

DraftJS er et React-rammeverk som for å sette opp en rik tekst editor. Dette rammeverket implementerer vanlige funksjoner som kursiv, fet tekst, overskrifter og bildeopplastning.[26]

DraftJS-wysiwyg

DraftJS-wysiwyg er en WYSIWYG-editor som er bygd på DraftJS. Den implementerer en modulær verktøylinje med knapper for å utføre vanlige WYSIWIG-funksjoner.

React Beautiful Drag n Drop

Et rammeverk som gjør det mulig å bygge drag-n-drop-funksjonalitet med JavaScript på en nettside. [8]

Redux

Redux er et JavaScript bibliotek for å håndtere tilstanden til JavaScript applikasjoner. Redux er i korte trekk bygd opp av noen få moduler; Store, state, reducers, og actions.

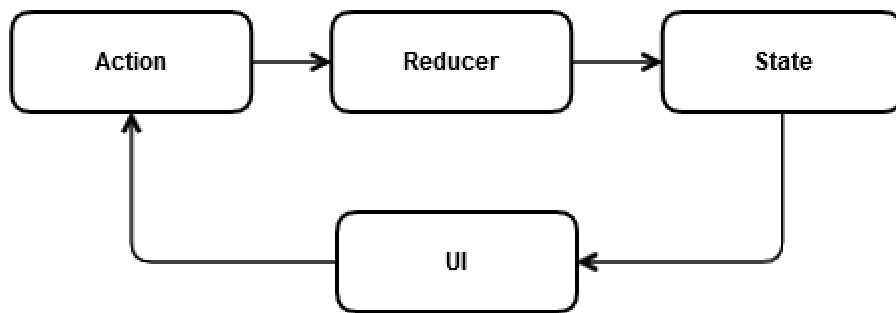
State er all data som er lagret i redux. State er immutable, som vil si at den ikke kan oppdateres direkte, men må oppdateres ved å lage en kopi med de nye endringene.

Store er objektet som tillater kommunikasjon mellom en applikasjon og state. Dette gjøres ved å tilby metoder for å registrere og fjerne lyttere, lese state, og en *dispatch* funksjon for å sende informasjon til state.

Actions er objekter som inneholder informasjon endringer som ønskes å gjøres i state. Actions blir sendt til state gjennom den nevnte *dispatch* funksjonen. Dette er den eneste måten en applikasjon kan sende informasjon til redux, og oppdatere state. En action består av et felt *type*, som sier hvilken type action det er, og en *payload*, som inneholder data.

En reducer innholder logikken for hvordan state skal oppdateres når en action forekommer.[61]

Et brukergrensesnitt kan dermed prate med redux ved å sende ut actions, og å reagere på endringer i state.



Figur 2.4: Flytdiagram som viser hvordan UI og Redux henger sammen.

React-Redux

React-Redux er React bindings for Redux. React-Redux har hooks, som gjør det mulig å benytte Redux-funksjonalitet i funksjonelle komponenter. Dette er positivt med tanke på best practices og ytelse. Ytelsen blir bedre siden man unngår å lage React klasser, som har mer overhead enn react komponenter.[59]

Redux-Toolkit

Redux-Toolkit er en kombinasjon av Redux og Redux-middlewares. Disse er satt sammen for å forenkle bruk og redusere mengden boilerplate-kode for Redux.

Redux-thunk

Redux-Thunk er et redux-middleware som utvider de asynkrone egenskapene til Redux. Redux-Thunk gjør det mulig å gjøre håndtere nettverkskall på en elegant måte.[62]

Axios

Et JavaScript-bibliotek som tilbyr en rekke sofistikerte verktøy for å administrere og manipulere HTTP-requests.[35]

Styled Components

Styled Components er et JavaScript bibliotek som gjør det mulig å legge CSS-styling i JavaScript-filer. Dette kalles CSS-in-JS.[19]

Material-UI og Material Design

Material-UI er et React bibliotek for å lage brukergrensesnitt som følger Google sine Material Design retningslinjer. Material Design er et designspesifikasjon utviklet av Google.[40] [39]

Tailwind

Tailwind er et lav-nivå CSS rammeverk [4] med fokus på små nytte-funksjoner for å lage mer tilpassede brukergrensesnitt.

Bootstrap

Bootstrap er et CSS rammeverk med fokus på ferdigbygde temaer og komponenter. [38]

React-Router

React-Router er et React bibliotek for å håndtere routing mellom React og nettleseren. React-Router er et bindeledd mellom HTML5 browser API[32] og web applikasjonen. [60]

2.13.12 Memoization

Memoization er en optimaliseringsteknikk der resultatet fra dyre funksjonskall lagres og returneres når samme funksjon senere kalles med samme input. [18]

2.13.13 Callback

En callback forklares av MDN Web Docs som en funksjon passert til en annen funksjon som argument, som så blir kalt inne i den ytre funksjonen for å fullføre en rutine eller handling". [2]

2.13.14 Profiler

En profiler er et verktøy som brukes for å måle ytelsen av et dataprogram.

2.14 Testing

I dette underkapittelet skal et par teknikker og teknologier for Automatisk testing beskrives, her blir det gjennomgang av Integration testing, unit-testing og Conformance testing.

2.14.1 Unit Testing

En teknikk for å teste den laveste mulige enheten i et system, der en enhet i objekt-orientert programmering vil være en metode. Målet med denne testen er å validere at hver enhet gjør som den skal, og vil alltid være det laveste nivået i testing.

2.14.2 Conformance Testing

Conformance testing er en teknikk for å teste at noe samsvarer med et sett med krav eller spesifikasjoner [3].

2.14.3 Integration Testing

Integration-testing er en teknikk for å teste funksjoner mellom flere moduler i et software-system. Der man ønsker å teste store deler av et system om gangen, i motsetning til for eksempel unit-testing der man tester på lavest nivå om gangen.

Kapittel 3

Metoder

I dette kapittelet skal det gjennomgås hvilke design-prinsipp, teknologier og verktøy som er brukt for å løse oppgaven.

3.1 Programvare og Verktøy

Denne listen gir en oversikt over verktøy som er brukt i oppgaven for å løse enkelte utfordringer eller som hjelpeverktøy for å bedre prosjektets gang.

- Gliffy - Et verktøy for å skissere diagram.
- Visual Studio Code - En text-editor med avanserte tillegsfunksjoner.
- JetBrains Rider - En utviklings-IDE.
- Azure DevOps - Et omfattende verktøy brukt til både Scrum-board og Pipelining.
- Git - Verktøy for versjonskontroll.
- Discord - Et VoIP-program med mulighet til skjermdeling.
- Teams - Et VoIP-program med mulighet til videomøte.
- Redux Devtools - En extension til nettleser for å debugge Redux state management
- React Developer Tools - en extension til nettleser for å debugge React
- BibItNow! - en extension til nettleser for kildehenvisning på nett.

3.1.1 Postman

Postman ble brukt for å strukturere, teste og dokumentere kommunikasjon mellom back- og front-end. Postman gjør det mulig å raskt og enkelt få testet oppførsel og responsen til backend med ulike HTTP forespørsler. Dette reduserer antall feil som kan oppstå ved implementasjon i frontend.

Alle endepunkt ble samlet og dokumentert i dette verktøyet. Dokumentasjonen dekker alle mulige responser for alle endepunkter. Postman har funksjonalitet for å generere en oversikt over alle samlede forespørsler. Denne oversikten kan redigeres om til et dokument som danner en oversikt over hele API-en.

3.2 Prosjektstyring

Dette underkapittelet beskriver alle metoder som er brukt i løpet av oppgaven for å øke kvaliteten på samarbeidet. Her blir det lagt frem hvordan Code Reviews ble gjennomført, et par hjelpemidler til pull-requests og smidig utvikling med Scrum.

3.2.1 Pull Requests

Når en utvikler er ferdig med en oppgave opprettes det en pull request 2.1. Når en pull request blir opprettet vil den gå gjennom automatiserte tester. Videre vil pull requestem gå gjennom code review 2.2.

3.2.2 Code Review

All kode gikk gjennom et code review 2.2 hvor hvert gruppemedlem måtte gå over pull requesten og godkjenne den. Om et gruppemedlem ønsket mer utdypning i pull requesten ble det lagd kommentarer hvor det ble stilt spørsmål. Utvikleren som lagde pull requesten vil så svare på spørsmål og ønsker fra gruppen. Her oppstår det diskusjoner angående implementasjon. Når hele gruppen har godkjent et code review, fusjoneres koden inn i utviklings-branchen.

Platformen for disse code reviewene var GitHub, der det også ble brukt støtteverktøy for å øke effektiviteten i hver code review. Verktøyene som ble brukt var DeepScan og Mergify.

DeepScan

DeepScan 2.13.2 ble satt opp til å automatisk kjøres ved hver pull request, for hjelpe med å opprettholde kvalitet på koden. Gruppen brukte kun den vanlige konfigurasjonen av DeepScan.

Mergify

Mergify 2.13.2 ble konfigurert til å automatisk merge en pull request, og slette branchen, når endringene var godkjent av hele gruppen, og all automatisk testing var ferdig og uten feil.

3.2.3 Smidig Utviklingsmetodikk

Dette underkapittelet beskriver hvilke metoder som ble brukt for å oppnå en smidig-utviklingsprosess i denne oppgaven. Alle metoder og verktøy for å oppnå dette beskrives videre i dette underkapittelet.

Scrum ble valgt som utviklingsmetodikk for dette prosjektet. Kort foltalt ble denne metodikken benyttet på grunn av måten arbeid blir organisert i Scrum, samtidig som at det å jobbe i intervaller passet settingen utviklingsoppgave i bachelorprosjekt. For å få en fullstendig oversikt over hvorfor denne ble valgt sei forprosjektrapport, kap 5.3.1 Utviklingsmetodikk i vedlegg D.

Sprint-review

I sprint-reviews 2.3.4 presenterte gruppen hvordan den siste sprinten hadde gått foran oppdragsgiver. Under dette møte får oppdragsgiver innsikt i hva som ble fullført, om noe ikke ble fullført og hvorfor. Møte inkluderer og som regel en demonstrasjon av de nyeste funksjonene. Oppdragsgiver kan så gi tilbakemelding og korrigere om oppgaver ikke ble som forventet. På dette viset får oppdragsgiver mulighet til å forme produktet.

Sprint-planning

Sprint-planning møter foregikk rett etter sprint-review. Her får oppdragsgiver velge hvilke arbeidsoppgaver gruppen skal fullføre til neste sprint-review. Alle arbeidsoppgaver var ferdig estimerte via planning-poker 2.3.6. Dermed hadde gruppen og oppdragsgiver innsikt i hvor mye arbeid som blir lagt i hver sprint.

Arbeidsmengden ble hver sprint estimert relativt til resultatet av forrige sprint. Unntak oppsto der det måtte tas høyde for oppdagelse av bugs fra brukertesting og rapportskriving.

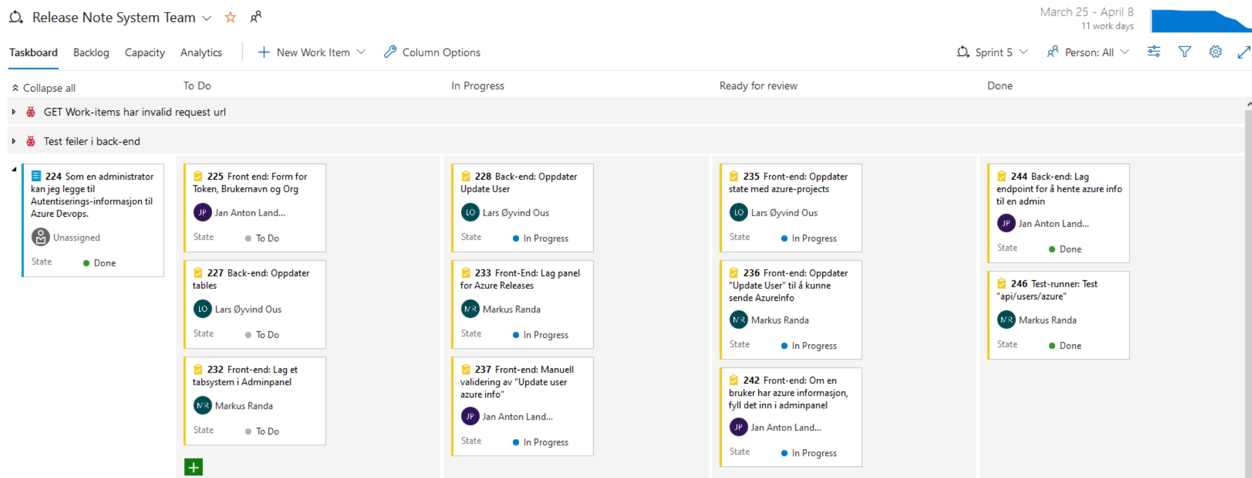
Alle møter mellom gruppen og oppdragsgiver ble dokumentert. Møtene ble dokumentert ved hjelp av to dokumenter. En røff kladd ble fortløpende skrevet i løpet av møte, og en finskrevet versjon som blir dannet etter møte. Alle finskrevne versjoner kan ses her i vedlegg K

Sprint-retrospectives

Etter at sprint-review og sprint-planning var fullført, utførte gruppen et sprint-retrospective 2.3.4. Her skrev gruppen et dokument hvor de oppsummerte hvordan den gjennomførte sprinten gikk, med mål om å få frem positive og negative aspekter. Alle sprint retrospectives kan finnes i vedlegg I

Azure DevOps

Gruppen valgte å bruke de samme verktøyene som oppdragsgiver for å gjøre seg kjent med verktøyene som systemet må samarbeide og integreres med. Azure DevOps inneholder blant annet et scrum-board 2.3.2. I scrumboardet oppretter gruppen task som er knyttet til en user story, videre vil et gruppemedlem velge en task og arbeide med den til den er klar til review via pull requests. Azure tilbyr også Azure Pipelines som blir i mer detalj beskrevet under testing kapitelene, se 3.7.1.



Figur 3.1: Eksempel på bruk av Azure DevOps scrum-board.

3.3 Planlegging

Dette underkapittelet gjennomgås det verktøy som ble brukt for utvikle brukergrensesnittet, program-flyt og tids-estimering.

3.3.1 Wireframes

For å planlegge hvordan brukergrensesnittet til nettsiden skulle se ut, ble alle sidene først tegnet i tegneprogrammet *Gliffy*, for så å bli implementert. En av årsaken til at denne teknikken ble benyttet, er fordi det tar mye kortere tid å lage en wireframe enn å utvikle et ferdig brukergrensesnitt. En annen årsak som bygger på forrige argument, er at det samtidig blir mulig å kjøre diskusjoner angående utseende før implementasjonen.

3.3.2 Gantt-diagram

Det ble opprettet et Gantt-diagram se vedlegg L, som inneholder en oversikt over samlinger av oppgaver som ble brukt til å estimere hvor lang tid oppgaven i sin helhet skulle ta. Disse samlingene består av større funksjoner slik som *Text editor*, som gir en veldig abstrahert og lett forklaring på hva som skulle gjøres i x uker. Andre ting som også er inkludert i diagrammet er *Brukertesting* og *Rapportskrivning*, der dette var noen store moment som også tok såpass mye tid at de ble inkludert i diagrammet.

3.4 Prosjektarkitektur

Prosjektet skal som nevnt bestå av en web applikasjon med et REST API, med grunnlag i prinsippet trelags-arkitektur, altså et presentasjons, business, og data-lag. De overordnede teknologiene i hvert lag er valgt ut fra ønske fra oppdragsgiver, men ved hvordan disse teknologiene brukes er det gitt nærmest full frihet. Dette

inkluderer implementasjon, og muligheter for bruk av ytterlige
dependencies. Hvordan hvert lag i systemet er bygd opp skal beskrives i denne seksjonen.

3.4.1 Presentasjons-laget

Presentasjons-laget, eller front-end, av systemet er som sagt bygd i React. React er et rammeverk som er laget for å bygge brukergrensesnitt og ikke noe mer. Mange funksjoner krever derfor enten egne løsninger, eller eksterne bibliotek. Nedenfor beskrives det hvilke funksjoner i React, samt eksterne bibliotek som utgjør arkitekturen.

Funksjonelle Komponenter og Hooks

Av de to forskjellige måtene å skrive React komponenter på 2.13.10, ble det brukt funksjonelle komponenter. Som et resultat brukes også hooks 2.13.10 systemet.

Optimalisering Av Ytelse

I de fleste tilfeller er React et god nok optimisert til at en ikke trenger å tenke på ytelse. Der dette likevel skulle være nødvendig, finnes det flere vertøy i React for å profilere og fikse ting som går ut over ytelsen. De viktigste verktøyene som ble tatt i bruk i løpet av utviklingen var Reacts *Profiler* 2.13.14, samt hooks for å bruke *memoization* 2.13.12.

For å profilere nøyaktig hva som bør optimaliseres, ble Reacts Profiler brukt. På grunn av Reacts iboende gode ytelse, er det tatt en profiler først, optimalisere senere –tilnærming. Det er først og fremst desginet på koden som er viktigst. Om det skulle oppstå problemer med ytelse, ble kilden til dette identifisert med profiling, og deretter vurdert å implementere en fiks.

Memoization er en vitkig teknikk for å optimalisere i React. Dette er på grunn av hvordan React holder brukergrensesnittet oppdatert; hver gang tilstanden til et komponent oppdateres, må den komponenten, samt alle underkomponenter gjengis på nytt. Om det foregår kostbare beregninger i ett eller flere komponenter, vil dette gå ut over ytelsen til resten av applikasjonen. Metoden brukt for å løse dette i prosjektet er memoization som gjør dette ved hjelp av `useMemo` og `useCallback` hooks. Disse hooks funksjonene lager respektivt en memoized verdi og callback funksjon.

Verktøy For Oppsett

- **Create-react-app**

Første oppsett av applikasjonen ble gjort ved hjelp av Create-React-App, som enkelt setter opp et prosjekt med moderne bygningsverktøy. Blant verktøyene som inkluderes er Babel, Webpack og ESLint. I tillegg inkluderer Create-React-App dokumentasjon som beskriver stegene videre i utvikling.

Verktøy For Tilstands-Behandling

- **Redux**
- **Redux-Toolkit**
- **Redux-Thunk**

Ved lagring av data i applikasjonen ble det bestemt ønskelig å opprettholde to prinsipper: *Single source of truth* og *Seperation of concerns*. Data som hentes inn skal være oppbevart sentralt, og nettverks- og lagringslogikk skal ikke blandes inn med UI-kode. Der det er behov for å lagre på komponent-spesifikk tilstand som for eksempel i en form, så brukes det lokal state.

På grunn av en-veis dataflyen til React 2.13.10, kan det lønne seg å bruke en sentral form for tilstandsbehandling, spesielt i større applikasjoner der flere komponenter avhenger av den samme dataen.

Det ble valgt å bruke Redux 2.13.11 for å løse dette problemet. Vanlige alternativer til Redux inkluderer Flux og MobX, men valget falt på Redux grunnet dets overveldende popularitet og store økosystem. I tillegg er det brukt Redux-Toolkit 2.13.11 for å redusere boilerplate, og Redux-Thunk 2.13.11 for å hjelpe med asynkrone handlinger.

Måten disse verktøyene ble satt opp for å opprettholde de nevnte prinsippene, var som følger: Alle API-kall ble implementert med thunk, til å asynkront dispatche *Pending*, *Error*, eller *Success* actions, om nettverkskallet respektivt er underveis, eller har returnert med feil eller suksess. Dette blir da det ene aksesspunktet applikasjonen har til APIen, og den globale tilstanden blir *single source of truth* for den resulterende dataen fra disse kallene.

Verktøy For Nettverkskall

Axios er brukt til alle nettverkskall. Axios ble valgt over JavaScripts innebygde *fetch* API på grunn av enklere API, og flere tilleggsfunksjoner som forenkler bruk. Nettverkskall i Axios utføres gjennom et *Axios* objekt. Dette objektet kan konfigureres etter ønske, blant annet til sette default headers på nettverkskall, sette en *base URL*, og mer. Det kan også settes opp *interceptors*. Dette gjør det mulig å stanse *requests* og *responses*, og anvende en funksjon før de sendes videre[11]. Axios objektet kan brukes både som en global instans, eller det kan skapes lokale instanser. Dette gjør det mulig å bruke forskjellige instanser der det kreves ulik konfigurasjon.

Verktøy For Navigasjon

React-Router

Metoden brukt for å navigere mellom de forskjellige sidene i nettsiden, var ved å implementere en react-router 2.13.11. Måten denne routeren er implementert på, er at ved endring av url, så byttes visnings-komponent til en predefinert komponent. For å konfigurere denne ruteren opprettes det en komponent som holder på en liste over kartlagte adresser mot komponenter.

Verktøy For Styling

- **Material-UI**
- **Styled-components**

Det var ønskelig lage en både konsistent og vakker stil i applikasjonen. Det var også et krav at applikasjonen skulle være mobilvennlig. Gruppen hadde derimot tidligere erfart at å oppfylle disse kravene kan være en tidskrevende, og hovedfokuset skulle være på selve funksjonaliteten. For å unngå å finne opp hjulet på nytt", ble det diskutert to forskjellige alternativer; CSS-rammeverk eller komponent-bibliotek. På den ene siden vil et rent CSS-bibliotek som Bootstrap 2.13.11 eller Tailwind 2.13.11 ha et mindre fotavtrykk, både på størrelse og ytelse. Dette vil også gi større kontroll over alle aspektene ved stylingen. På den andre siden vil et komponent-bibliotek være mer integrert med React, som gir en mer standardisert utvikler-opplevelse. Det vil også ha mulighet for mer funksjonalitet ut av boksen, derav spart tid. Det ble derfor valgt å bruke komponent-biblioteket Material-UI 2.13.11.

Selv med ferdigbygde komponenter fra et komponent-bibliotek, vil det i mange tilfeller også være nødvendig med mer tilpasset CSS. Styled-components 2.13.11 ble derfor tatt i bruk for å skrive CSS i applikasjonen. Dette biblioteket lar deg skrive CSS direkte i et React-komponent. På denne måten unggår man lange CSS-filer, og hver komponent blir også mer selvstendig.

Testing

- **Jest**
- **React-Testing-Library**

For å kjøre tester er det brukt Jest og React-Testing-Library.

Jest har testing-rammeverk, som vil si at den er ansvarlig for alle funksjoner relatert til å skrive og kjøre tester i prosjektet. Jest var et naturlig valg, da det er inkludert i *create-react-app*, og ikke krever oppsett.

React-testing-library er et bibliotek for å teste react komponenter. Dette biblioteket eksponerer et API som etterligner måten en ekte bruker ville interaktert med siden på. Testene blir derfor mer sentrert rundt hva som kan sees på skjermen, og bryr seg ikke om hvordan komponenter er implementert. Dette biblioteket er også ett av to av

create-react-app sine anbefalte test-biblioteker. Alternativet er shallow rendering med *enzyme*. Denne tilnærmingen skiller seg ved at komponenter blir testet isolert fra komponent-treet.

3.4.2 Business og data-laget

Business og data-laget til prosjektet utgjør backenden til prosjektet, og skal eksponere et REST API som skal brukes av front-enden. REST API-et ble bygd i ASP.NET Core, med postgres som database.

Services

Alle klasser på applikasjonsserveren som har med å databehandling å gjøre, ble definert som hver sin service 2.5.2. Årsaken til at gruppen valgte å gjøre dette er på grunn av fordelene man får med tanke på dependency injection 2.5.3 som igjen gir løsere koblet kode.

Unit of Work

Metoden som ble brukt for å synkronisere alle handlinger mot databasen ble gjennomført med design mønsterert unit of work 2.5.5. Måten dette ble implementert på var gjennom constructor injection 2.5.4, slik at alle services som hadde bruk for å persiste data bare inkluderte Unit of Work-klassen i konstruktøren sin.

Auto-mapper

Metoden som ble brukt for å mappe felt fra en Resource-klasse til en Modell-klasse var ved hjelp av AutoMapper som blir forklart i kap 2.5.6. Dette ble gjort ved å implementere AutoMapper som en service også legge inn konfigurasjon for hver enkelt mapping.

Controller-klasse

Ansvaret til denne type klasse var å representere logikken for et spesifikt endepunkt. Med dette menes det altså at et endepunkt er adgangen satt for en entitet. I denne klassen ble det definert metoder for å hente ut eller manipulere data. I kommunikasjons-hierarkiet så kommer denne klassen først. En annen oppgave denne klassen har er å mappe fra Modell til Resource slik at kun resources blir sendt i responses.

I denne klassen blir også metodene for http 2.7.1 benyttet for requests motatt av klassen, denne konvenensjonen er implemenrt på følgende vis:

- **GET** Denne metoden brukes når data skal hentes ut.
- **PUT** Denne metoden brukes når data skal oppdateres.
- **POST** Denne metoden brukes når data skal opprettes.
- **DELETE** Denne metoden brukes når data skal slettes.

En annen konvensjon som benyttes for klassen er http-statuskoder 2.7.2, der konvensjonen som ble benyttet er som følger:

- **200** Ble benyttet for vellykket request.
- **400** Ble benytett til requests som hadde manglende eller feil innhold.
- **204** Ble benyttet hvis innhold ikke ble funnet.
- **401** Ble benyttet ved mislykket autorisering.

Den siste konvensjonen som gruppen bestemte seg for å bruke var hvordan query-parameters 2.7.3 og path-parameters 2.7.4 skulle brukes for hvert endepunkt. Denne konvensjonen går som følger:

- **Query Param** Benyttes til oppgaver som filtrering, sortering og andre tilfeller der det er behov for å si noe mer om requesten.
- **Path Param** Benyttes for å identifisere "hva". Hvis man for eksempel skal hente ut et produkt så vil man legge til ID-en til produktet som et path-parameter.

Service-klasse

Ansvaret til denne type klasse er å fungere som en tjeneste for en spesifikk entitet i systemet. Denne klassen blir referert til av en Controller av samme entitet. Jobben til denne klassen er i all hovedsak å fungere som et bindeledd mellom Controller og Repository. Det klassen skal avgjøre er hvilket resultat som returnes til Controller og hvilken data som skal hentes eller manipuleres i Repository. En mulighet denne klassen har er å bruke services og repositories for andre entiteter i tilfeller der det er behov for å kombinere data. I noen tilfeller blir også denne klassen nødt til å endre på data, dette kan for eksempel være hvis et modell-objekt skal returneres som en resource, da blir en mapper 3.4.2 brukt.

Repository-klasse

Ansvaret til denne type klasse er å håndtere all direkte kommunikasjon mellom applikasjonsserveren og databasen. Måten denne klassen forholder seg til service-klassene er ved å tilby metoder for uthenting og manipulering av data.

Modell-klasse

Ansvaret til denne type klasse er å opptre som entitet i systemet, der en modell blir utgangspunktet for en tabell. Navnet på denne klassen og feltene definerer hvordan tabellen i databasen blir definert. Dette blir utdypet i kap 3.4.2.

Resource-klasse

Ansvaret til denne type klasse er å opptre som en alternativ utgave av en modell som består av færre eller flere felt. En av måtene denne klasse-typen blir brukt på er når det er behov for å sende data til front-end, da vil bare de feltene som kan eksponeres for front-end blir inkludert. En annen situasjon der denne klasse-typen blir brukt er når innkommende JSON-objekt skal begrenses, så kan denne klasse-typen settes opp med felt man forventer å motta på et endepunkt. En metode som også er viktig å nevne er at denne klassen kan inkludere regler for sine felter, som på et senere tidspunkt brukes til validering.

Response-klasse

Ansvaret til denne type klasse er å opprtre som en mer utdypende response, der klassen er knyttet opp mot en entitet. Denne klasse-typen gir en struktur som gjør det oversiktlig å inkludere mer data enn en vanlig response, der det kan bli behov for å gi mer tilbakemelding enn bare et enkelt objekt eller beskjed.

Startup-klassen

I startup skjer det hovedsaklig to store ting: Middlewares og dependency injection blir konfigurert. Dette skjer etter standard ASP.NET Core praksis, gjennom to metoder: `Configure` og `ConfigureService`. I `ConfigureService` registreres alle dependencies som skal brukes til dependency injection-rammeverket. Dette inkluderer alle Controller, Service, og Repository klasser, database, autentisering og autorisering, og diverse andre hjelpeklasser. I `Configure` registreres all middleware som skal brukes. I dette prosjetet brukes følgende middleware: Hosting av statiske filer, routing, autentisering og autorisering, og endepunkt i form av Controllers.

AppDbContext-klassen

Ansvaret til denne klassen er å konfigurere og aksessere databasen. Klassen arver fra en klasse i EF-Core `DbContext` som enkelt forklart representerer en sesjon med databasen. Det som blir konfigurert i datbasen er oppsett av tabeller der Model-klasser blir brukt, default-verdier blir satt og alle andre konfigurasjoner som har med tabellen å gjøre.

Database-design

Metodene som ble brukt for designet på databasen består av en navnekonvensjon og noen teknikker for å håndtere forhold mellom forskjellige tabeller. Denne navnekonvensjonen og teknikkene ble implementert med rammeverket EF-Core 2.13.7. Navnekonvensjonen er ganske enkel, alle tabellene representerer en enkelt entitet, der navnet på tabellen blir entitetens navn i flertallsform. Teknikken som ble brukt for å håndtere forhold mellom tabellene var one-to-many, one-to-one og many-to-many, disse ble forklart i kap 2.8.5.

Måten gruppen gikk frem for å identifisere hvilke teknikker som skulle brukes hvor skal forklares videre. Det gruppen begynte med var å bygge opp datamodellen der det ble identifisert hva som skulle være entiter og hvilke re-

lasjoner disse har mellom seg. Måten en entitet ble identifisert på baserte seg på oppgaven som skulle løses og hvilken informasjon som skulle lagres på og behandles. Da ble det mulig for gruppen å begynne å definere en og en entitet med sine respektive felt.

Det andre gruppen måtte ta hensyn til i denne prosessen var hvordan entitetene forholdte seg til hverandre. Da måtte gruppen se på hva disse forholdene var, og teknikkene one-to-one, one-to-many og many-to-many ble brukt videre. For alle entiteter der man kunne si at entitet A kan ha flere relasjoner til entitet B, og entitet B kan ha flere relasjon til entitet A", ble det satt opp et many-to-many-forhold. For alle entiteter der man kunne si entitet A kan ha en relasjon til entitet B og entitet B kan ha flere relasjoner til Able det satt opp et one-to-many forhold. For alle entiteter der man kunne si at entitet A kan bare ha en relasjon til entitet B, og omvendt", så ble det satt opp et one-to-one-forhold.

Når navnene på alle entitene og forholdene mellom entitene var på plass, ble de implemenetert med EF-Core. Implementasjonen ble gjennomført på følgende måte:

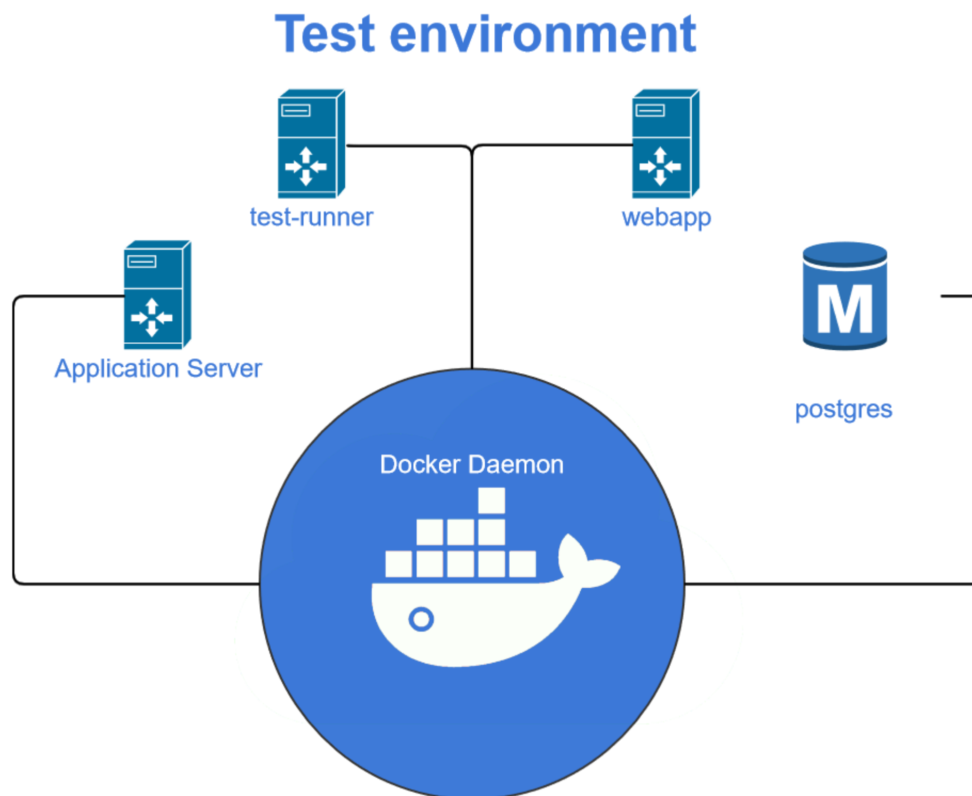
1. Alle entitetene ble opprettet som Modell-klasser med sin respektive felt.
2. Deretter ble Naming conventions se kap 2.13.7 brukt for å sette opp forholdene mellom entitene og tabell-navnene.
3. Til slutt ble Modellene lagt inn i konfigurasjonen til klassen `AppDBContext` se kap 3.4.2.
4. Når dette var på plass, så var det bare å kjøre oppe applikasjonen mot en tom database.

3.4.3 Mikrotjenester

En teknikk som ble benyttet for å sørge for rask testing, utvikling og deployment, var mikrotjenester. Her tok utviklingsteamet i bruk prinsippet om mikrotjenester 2.10, der hver tjeneste i systemet ble plassert i hver sin respektive container. De tjenestene utviklingsteamet valgte å inkludere i dette prosjektet var:

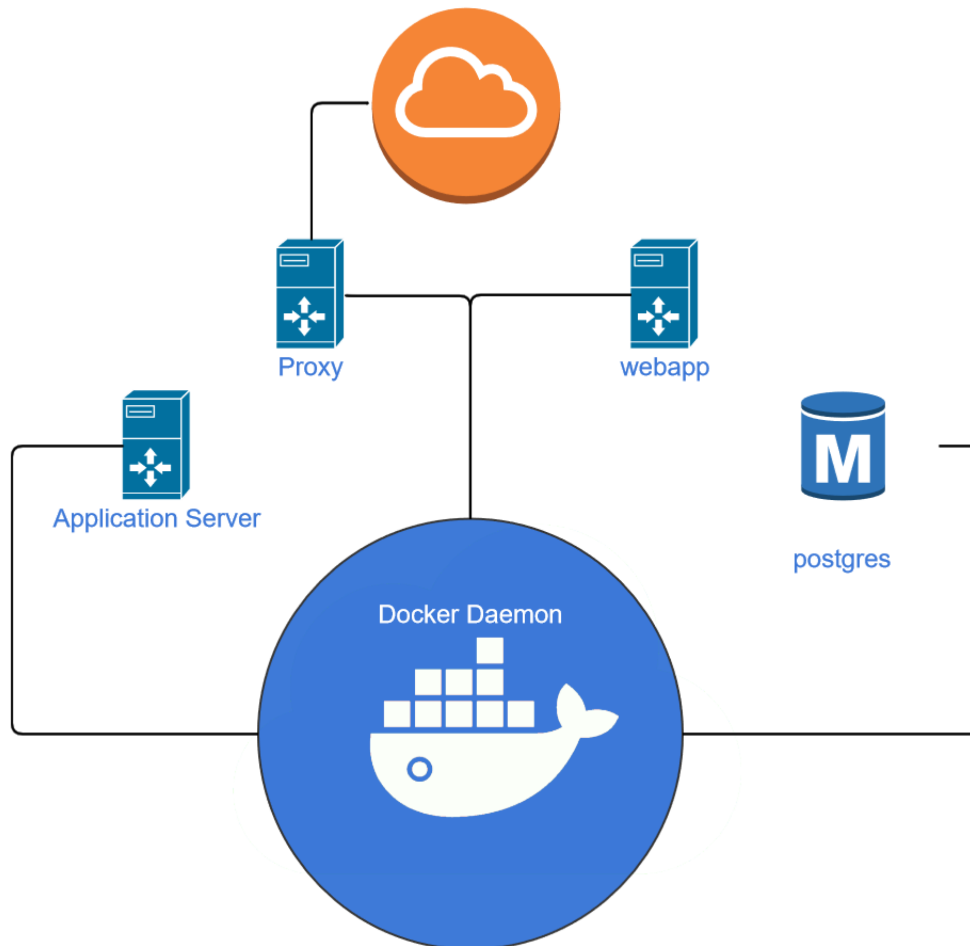
- application-server
- test-runner
- postgres
- webapi
- proxy

Her så utviklingsteamet at det var lønnsomt å benytte seg av Docker-Compose 2.10.2 for å veldig enkelt få opp tre forskjellige miljø.



Figur 3.2: Oversikt over mikrotjenester i test-miljø.

Fig. 3.2 viser et miljø brukt kun til å kjøre testing på services. Her foregår alt i et lukket docker-nettverk, der hovedsaklig container test-runner kjører HTTP-kall til applikasjonsserver og validerer resultater. Akkurat hvordan denne testingen foregår, vil bli tatt opp i subsection 3.7.2.



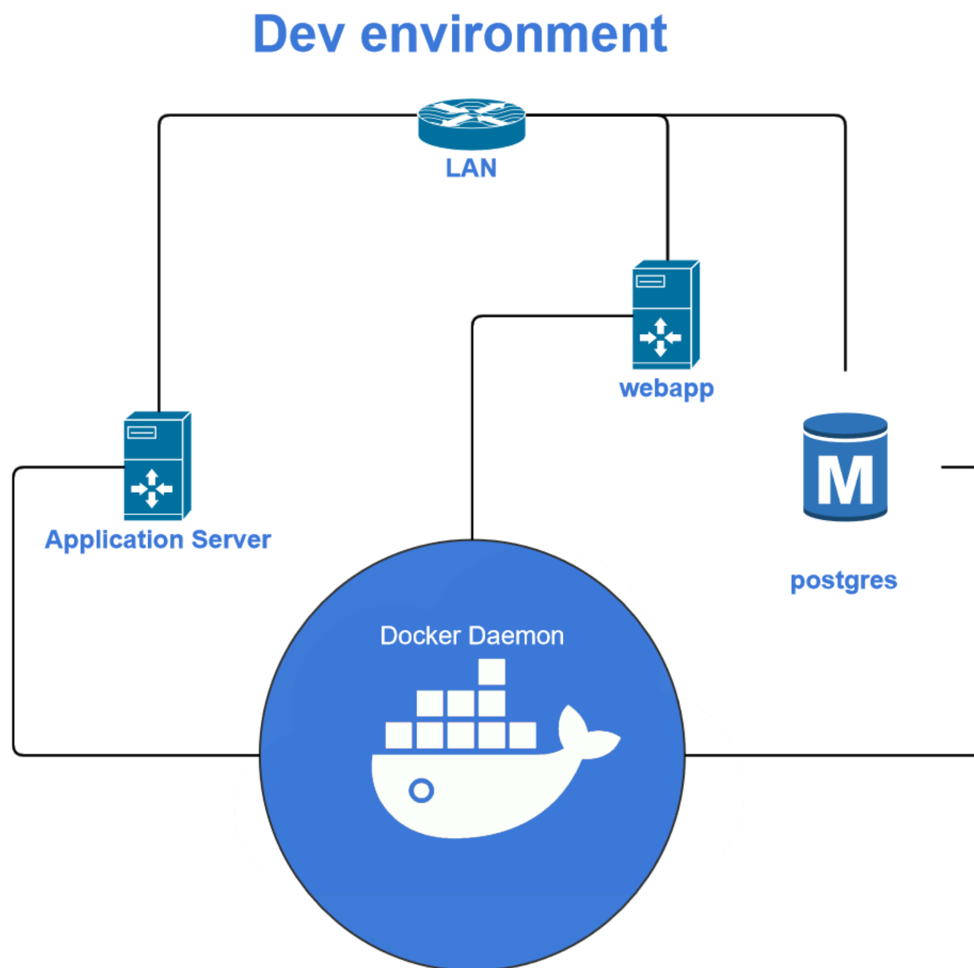
Figur 3.3: Oversikt over mikrotjenester i prod-miljø.

Fig 3.3 viser et miljø brukt som produksjonsmiljø. Her foregår alt i et nesten lukket docker-nettverk, der kun en proxy-container blir eksponert ut av docker-nettverket. Dette gjør at man enkelt kan sette opp et produksjonsmiljø der tilgangen begrenses av en proxy, og webapplikasjonen ikke trenger å benytte seg av porter når forskjellige tjenester skal aksesseres.

3.4.4 Reverse-proxy

En proxy-tjener ble konfigurert til å filtrere og videresende alle forespørsler fra klient-maskiner til en intern tjener. Denne proxyen er konfigurert som en reverse-proxy 2.9 som videresender forespørsler til den korrekte tjenertje-

nesten.



Figur 3.4: Oversikt over mikrotjenester i utviklings-miljø.

Miljøet vist i fig. 3.4 er kun ment til utvikling av systemet. Her er både applikasjonsserveren og webbapp-serveren tilgjengelig ut i nettverket. Årsaken til at flere containers er tilgjengelig i nettverket er fordi det benyttes verktøy som JetBrains Datagrip og JetBrains Rider debugger, til feilsøking i utviklingsprosessen.

3.5 Funksjonalitet

Det som gjennomgås i dette underkapittelet er en gjennomgang av alle betydelige bibliotek som ble benyttet for å oppnå mer spesialisert funksjonalitet.

3.5.1 Teksteditor

For redigering og lagring av rik tekst ble det brukt DraftJS 2.13.11. DraftJS ble valgt på grunn av dets store muligeter til å tilpasses til prosjektets behov. DraftJS oppfyller også kravet om WYSIWYG med hjelp fra DraftJS-wysiwyg 2.13.11. I tillegg er DraftJS veldig godt produksjonstestet, i og med at det brukes i som input i blant annet Facebooks kommentarfelt og Messenger.

3.5.2 Drag-and-drop

For å få mulighten til å kunne dra og slippe elementer på nettsiden, ble biblioteket til Atlassian react-beautiful-dnd se 2.13.11 benyttet. Dette er et mye anvendt JavaScript-bibliotek som gjør det mulig å bygge avanserte brukergrensesnitt der det finnes gode muligheter til å endre oppførselen til elementer.

Årsaken til at dette ble benyttet over alternativene var på grunn av det aktive samfunnet som bruker dette biblioteket, det betyr at det finnes mange eksempler på bruk og man vet at det er godt testet.

3.6 Sikkerhet

Det dette underkapittelet skal ta for seg er hvordan metoder som ble benyttet for sikre systemet mot misbruk der både metode for autentisering og autorisering blir gjennomgått.

3.6.1 Brukerautorisering

Måten Brukerautorisering ble implementert på var ved å bruke Stateless Autentication 2.6.2 med JWT 2.6.1. Det betyr at brukere av systemet logger på en gang med passord og brukernavn, og mottar en token som lagres i nettleser. Når token er lagret i nettleser, kan alle andre autoriserte handlinger i systemet autoriseres ved hjelp av token, slik at brukeren får en opplevelse av at han/henne er innlogget.

For å sjekke hvilke handlinger i systemet en spesfikk bruker har lov til å bruke, ble det implementert en rollebasert autoriseringssjekk. Det vil si at hver bruker har sin spesifikke rolle, som igjen benyttes for å vurdere om denne brukeren har tilgang til gitt handling. Dette rollebaserte autoriserings-systemet ble implementert med tanke på at det kan bli flere roller i fremtiden.

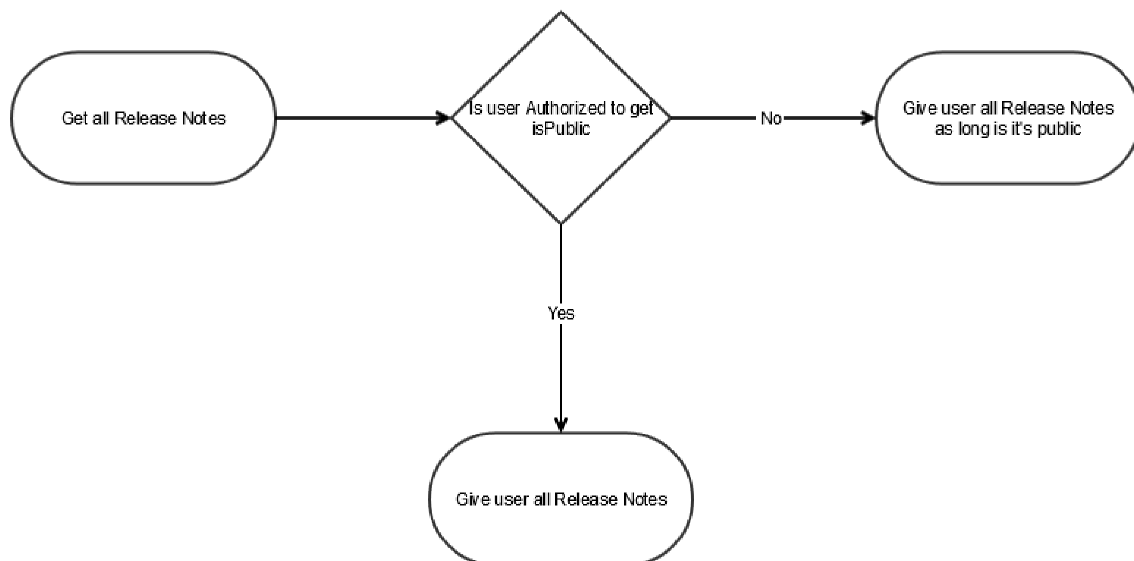
3.6.2 Dataautorisering

Som en utvidelse til autoriseringsjekk på endepunkt ble også Data Authorization 2.6.3 benyttet for å øke sikkerheten til systemet. Måten denne teknikken ble implementert på var ved å sette globale filterings-regler for spesifikke tabeller, slik at ved forespørsel kan en bruker bare få ut det som er tillat for akkurat den brukeren. Det som skulle sikres i dette systemet var all data som er flagget med den boolske verdien *isPublic*. Denne verdien symboliserer om database-entiteten er tilgjengelig for alle, eller kun administratorer. Eksempel på denne flyten finnes i fig 3.5.

p = Entitet er åpen for offentligheten

q = Bruker er administrator

p	q	$p \lor q$
0	1	1
1	0	1
1	1	1
0	0	0



Figur 3.5: Flyt-diagram som viser eksempel på uthenting av Release Note med data-autorisering.

En fordel med denne typen autorisering, er at det stilles et veldig lavt krav til at utvikler husker å sikre alle tilganger når man ved flere anledninger aksesserer en gitt tabell. En ulempe vil være at denne sjekken kjøres hver gang tabellen aksesseres, som gir lavere ytelse.

3.7 Testing

Det som skal forklares i dette underkapittelet er hvilke metoder som ble benyttet ved testing. For å sikre at alle nye bidrag til prosjekts kildekode ikke ødela eksisterende funksjonalitet, eller ikke fungerte som forventet, ble det benyttet et par metoder for automatisk testing.

3.7.1 Azure Pipelines

Azure Pipelines 2.12.3 er brukt i prosjektet for å automatisk kjøre tester ved pull requests på GitHub. Måten dette gjøres er ved å først peke til repositoryet på GitHub der kildekoden ligger, og deretter inkludere en `azure-pipelines.yml` fil i roten av repositoryet. I denne defineres det stegvis hvilke oppgaver som skal utføres når pipelinen kjøres. Typene oppgaver som kjøres kan enten være *tasks* 2.12.3 eller skript. Det må også defineres hvilke betingelser som må møtes for at pipelinen skal kjøres.

3.7.2 Integration Testing

En av metodene for å automatisk teste koden i prosjektet var ved å benytte teknikken integration testing, med REST API som utgangspunkt. Det som ble gjort for å enkelt teste funksjonalitet på applikasjonsserveren, ble gjort med en ny container som sendte requests til applikasjonsserveren der responses ble testet opp mot assertions. Måten disse assertions ble definert på, var gjennom akseptansekravene satt per user story. En annen måte disse kravene ble satt på var gjennom erfaringer underveis der for eksempel en feilsituasjon hadde oppstått på grunn av manglende feilhåndtering.

3.7.3 Unit-testing

For å forsikre at Redux-logikk utførte forventete handlinger ble lagd unit-tester med Jest. Her ble det satt opp eventuelle mocks for å imitere server responser.

3.7.4 Conformance Testing

Conformance testing 2.14.2 er brukt i dette prosjektet for å teste kode der implementasjonen av et sett funksjoner kan være ulik, men utbyttet av funksjonen er forventet å følge et gjengående mønster. Dette er nyttig for redusert kodeduplisering, og å teste potensielt store mengder kode med liten innstats.

3.7.5 Brukertesting

Brukertesting ble anvendt for å sikre at produktet fungerte som forventent i bruk, og samsvarte med kundens ønsker og. Det ble utført flere runder med brukertesting, etter hvert som nye funksjoner og bug-fixes ble implementert.

Teknikken som ble benyttet for utføre brukertestinger var *Unmoderated Remote Usability Testing* 2.11.1. Måten denne teknikken ble implementert på var ved å gjøre en prototype av produktet tilgjengelig for test-subjektene, og

opprette et skjema se 3.6. Prototypen bestod av alle endringer gruppen anså som ferdige, og skjemaet ble fylt ut for å teste de samme endringene. Når både prototypen og skjema var på plass, så begynte oppdragsgiver å teste foreslåtte funksjoner i produktet.

Skjema for tilbakemelding på funksjoner

Her tenker vi at dere legger inn noe tilbakemelding for hver funksjon, men bugs kan dere rapportere direkte i Azure Boards. Her er det bare å kontakte oss på mail hvis dere mangler tilgang eller har spørsmål.

Funksjon	Kan bli bedre	User Story	Bugs
Eksempel	*1. Tok for lang tid før skjedde noe* *2. Ikke intuitivt plassert* *3. Dårlig tilbakemelding* *4. Her kan ingenting bli bedre!*	*1. X* *2. 0* *3. X* *4. X*	*https://dev.azure.com/ReleaseNoteSystem/Release%20Note%20System/_workitems/edit/218/*
Generelt			
Logg på			
Les en release			
Sorter releases			
Gå inn i adminpanel			
Opprett/redigere et produkt			
Opprett/redigere en bruker			
Opprett/rediger en release note			
Opprett/rediger en release			
Slette Release			

Figur 3.6: Her er oversikt over hva tilbakemeldingsskjemaet består av.

- **Funksjon** Hvilken funksjon som skulle testes, utvikler fyller ut.
- **Kan bli bedre** Tilbakemelding fra bruker, bruker fyller ut.
- **User Story** Oversikt om funksjonen har fått user story, utvikler fyller ut.
- **Bugs** Mulighet for bruker å opprette bug rett i Azure boards.

En av årsakene til at akkurat denne brukertesting-teknikken ble benyttet over andre teknikker som for eksempel *Guerilla Testing* 2.11.3, var på grunn av det begrensede antallet brukere som skulle bruke produktet, som i dette tilfellet begrenser seg til omkring tre personer. En annen årsak til at akkurat denne metoden ble brukt var på grunn av det lave kravet oppdragsgiver stilte til utseende, der nyttige funksjoner trumfet god utseende. Hvis utseende hadde vært enda viktigere i oppgaven, så kunne for eksempel en metode som *Lab Usability Testing* 2.11.2 vært en bedre metode. Styrken til *Lab Usability Testing* er den er litt mer vitenskapelig går frem for å forstå hvordan vår løsning fungerer for personene som skal benytte seg av systemet, men samtidig vil den metoden også ha påført gruppen mer arbeid.

Etter hver fullførte runde med testing ble det ut fra resultatene vurdert hvilken handling som skulle tas. Enten ble løsning på tilbakemeldingene implementert i samme sprint som brukertesting, eller så ble tilbakemelding drøftet med oppdragsgiver i neste sprint review, og deretter planlagt inn i en fremtidig sprint. Denne vurderingen ble gjort med hensyn til kompleksitet, tilgjengelig arbeidskraft og hvordan løsningen kunne ha blitt gjennomført.

Kapittel 4

Resultat

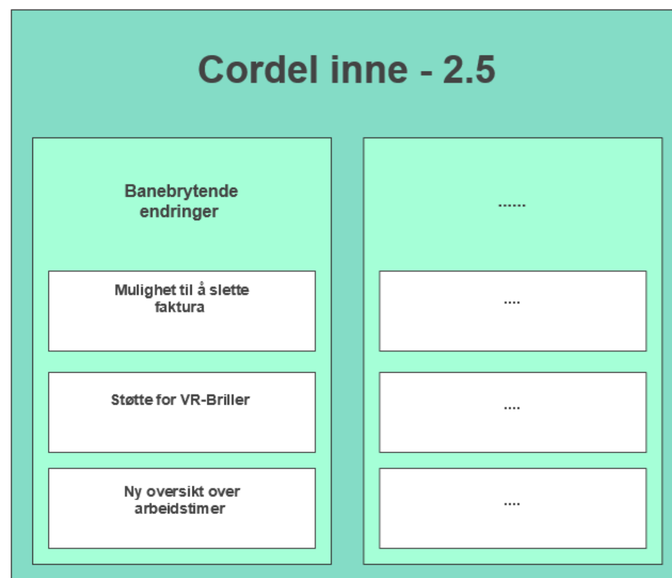
Dette kapittelet gjennomgår hvordan designprinsippene og teknologien beskrevet i kapittel 3 ble implementert og hvilke resultater de ga oppgaven. Implementasjonen inkluderer en oversikt front-end, back-end, brukergrensesnitt, funksjonalitet og testing.

4.1 Funksjonalitet

Dette avsnittet skal vise en oversikt over hvilke muligheter som finnes i det fullstendige systemet. For å få en fullstendig oversikt over absolutt alle funksjonene til nettsiden, med forklaring av bruk, så finnes dette i brukermanualen H.

4.1.1 Redigering av Releases & Release Notes

Mest sentralt i systemet står releases og release notes. Det er derfor viktig å forstå oppbygningen av disse, samt å forstå forholdet mellom entitetene ProductVersion, ReleaseNote, og Release.



Figur 4.1: Oversikt over sammenhengen mellom Release og Release Note

Dette forholdet er illustrert i figur 4.1. Nederst i hierarkiet vises en *Release Note*, som er et stykke tekst som doku-

menterer én implementert funksjon. Videre ser man en *Release*, som er bygd opp av flere release notes. En *Release* beskriver dermed alle nye funksjoner som er inkludert i en ny revisjon i et *Product*.

Videre i dette avsnittet skal det beskrives hvordan disse entitetene er integrert i systemet.

Release Notes

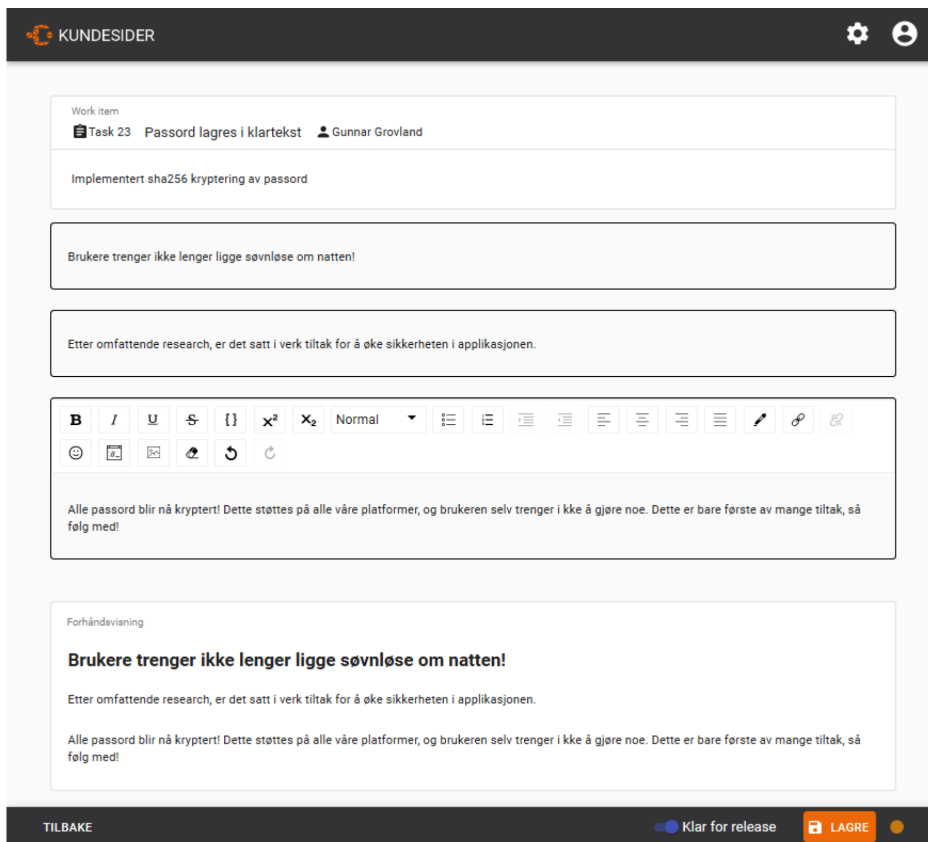
Innholdet i en Release Note består av en tittel, ingress, og en detaljert beskrivelse. Release Notes kan redigeres i systemet gjennom *ReleaseNoteEditor* 4.2 visningen. Her ser man felt for å redigere tittel, ingress og beskrivelse i midten. Beskrivelse-feltet har rik-tekst kapabilitet etter WYSIWYG prinsippet.

Øverst i editoren ligger en boks som inneholder informasjon fra en Work Item. Dette finnes for å hjelpe skriveren av en Release Note ved å gi kontekst til funksjonen som er implementert, samt hvem som er ansvarlig for Work Itemen.

Den nederste boksen i editoren er en live forhåndsvisning av innholdet i Release Noten.

Ved opplasting av bilder fra lokal maskin, vil blir disse automatisk lastet opp og lagret på API Serveren.

Verktøylinjen i bunnen av skjermen inneholder knappen for å gå tilbake til adminside, endre offentligheten til Release Noten, og lagre. Her finnes det også indikator som forteller om det finne ulagrede endringer i Release Noten.



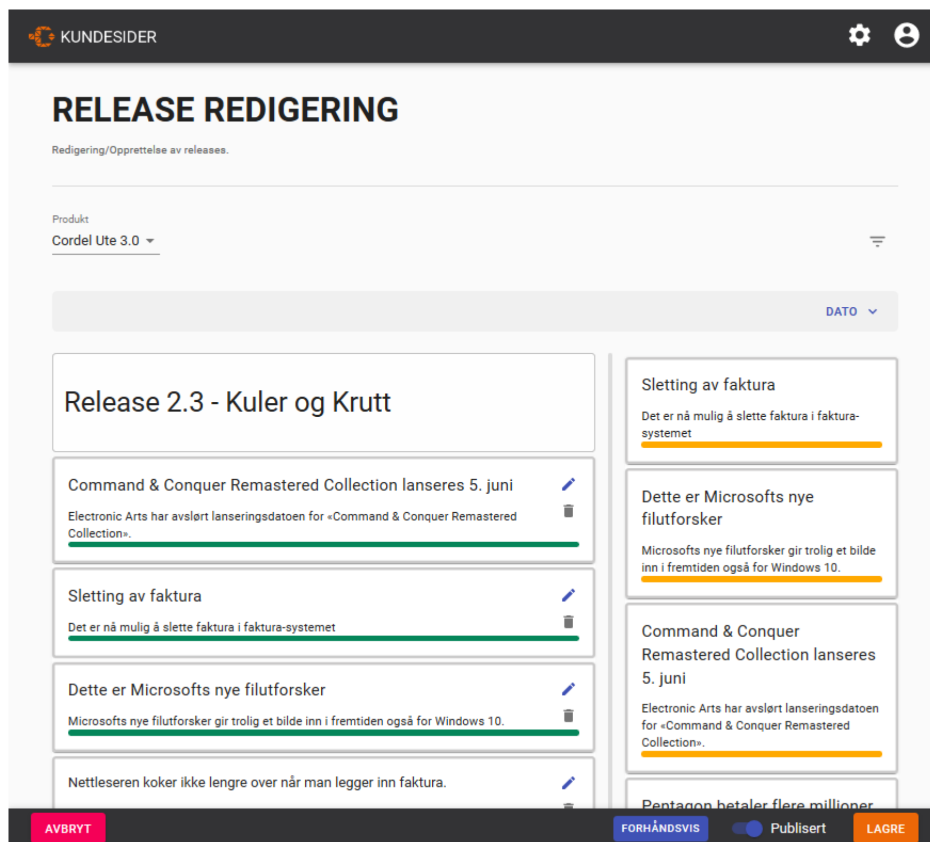
Figur 4.2: Release Note Editor

Release

En release består som sagt av flere release notes, og er knyttet til en produktversjon. Måten en administrator kan bygge opp releases er via ReleaseEditor 4.3, som er et drag and drop system, der det er mulig å dra fra en liste tilgjengelige Release Notes, over til en release. I tillegg finnes det felter for å skrive tittel og velge produkt til releasen.

Verktøylinjen i bunnen av skjermen inneholder knapper for å gå tilbake til adminside, forhåndsvise den fulle releasen, endre offentligheten til releasen, og lagre.

Verktøylinjen over kolonnene for Release og Release Notes inneholder en funksjon for filtrering av Release Notes, der det er mulig å filtere på dato innenfor et gitt intervall. Det er også mulig å editere Release Notes direkte i ReleaseEditor, der det finnes en blyant-knapp på ReleaseNotes i Release-kolonna som åpner opp en modal med fullstendige redigeringsmuligheter.

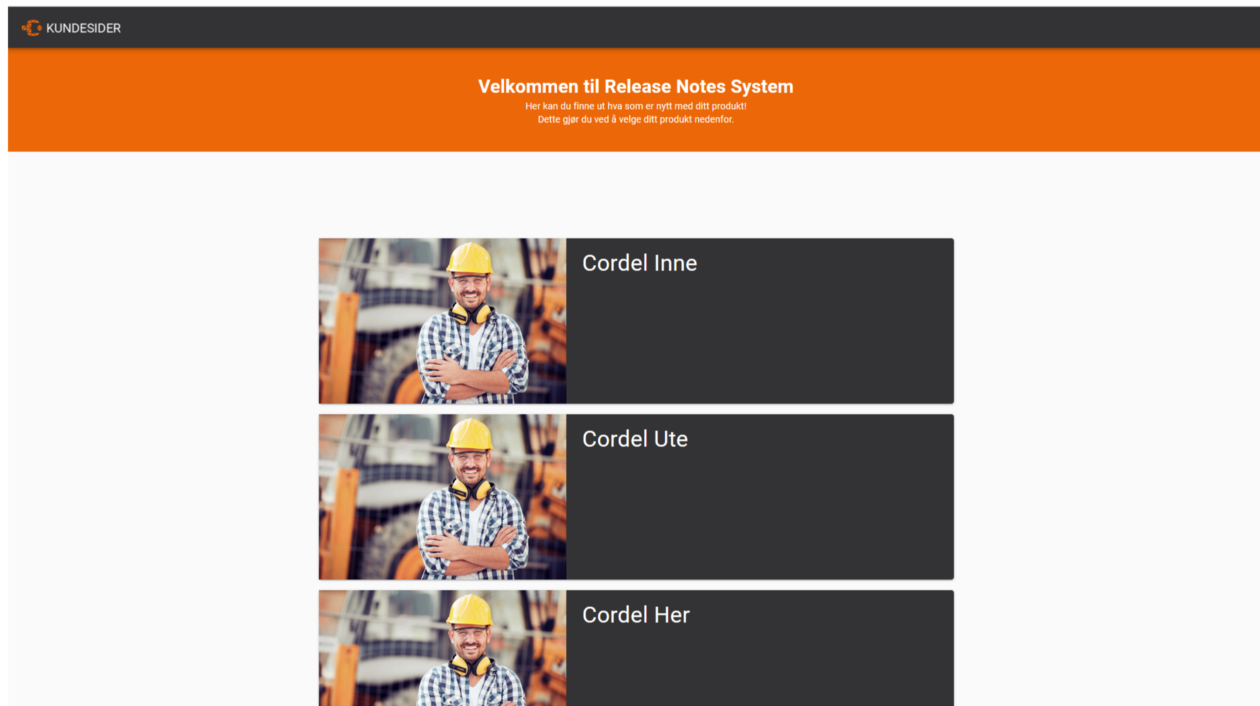


Figur 4.3: Release Editor

4.1.2 Offentlig side

Her er oversikten over funksjonene til den offentlige delen av nettsiden.

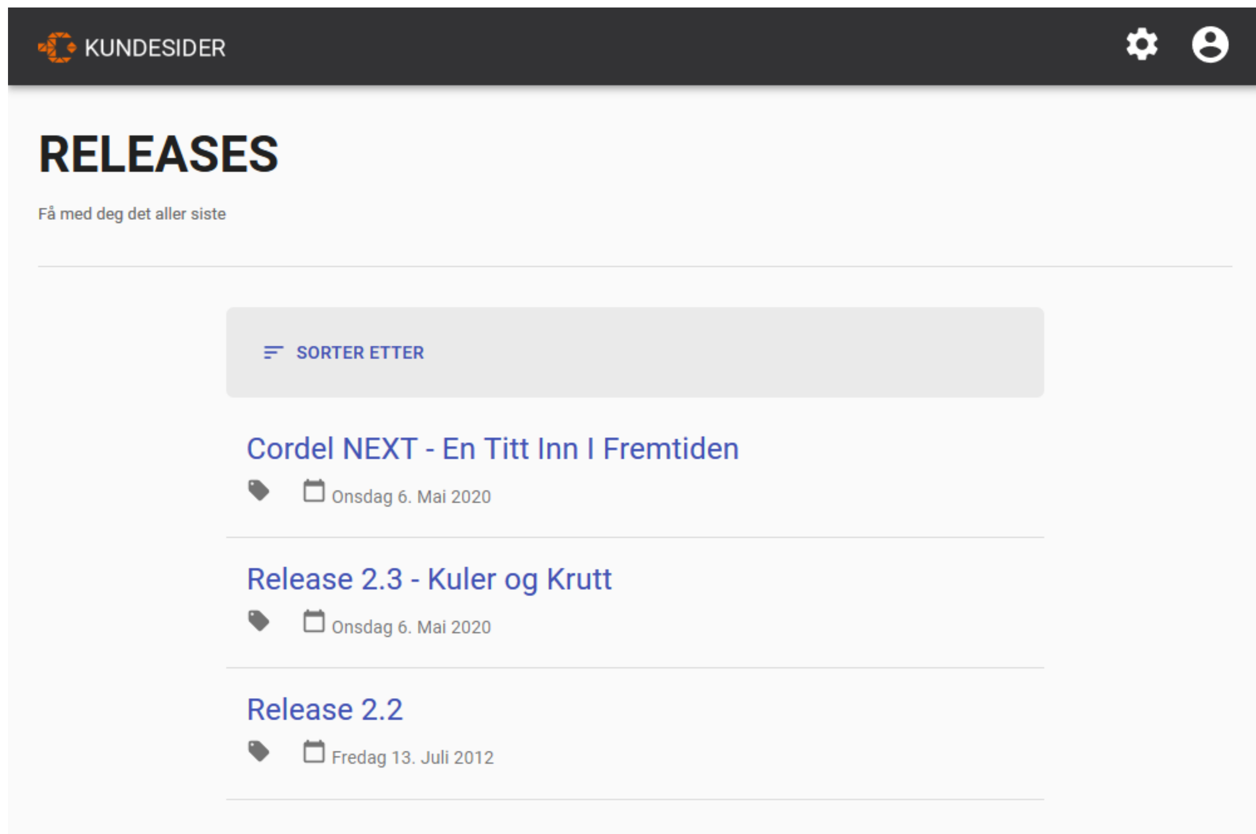
Forside



Figur 4.4: Utseende på forsiden til nettsiden.

Releases-side

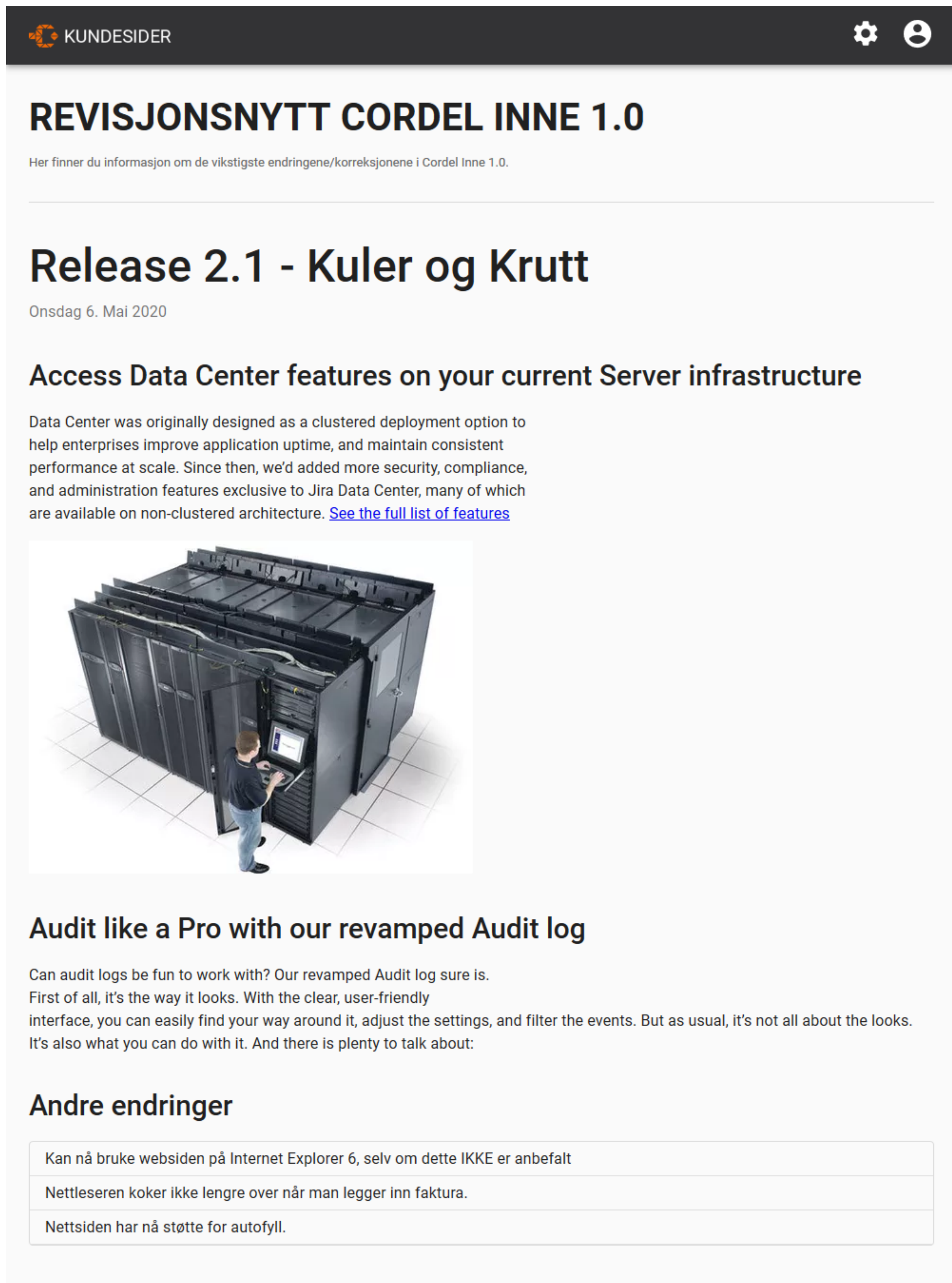
På denne siden kan man se alle releases som er tilgjengelige for valgt produkt. Eksempel på hvordan dette ser ut vises i fig. 4.5



Figur 4.5: Oversikt over alle releases tilgjengelig for valgt produkt.

Release-artikkel

En release kan leses på denne siden, som vises i fig. 4.6. Her vises alle release notes som er inkludert i en relesase, med skille mellom større og mindre endringer. Større endringer vises øverst på siden med tittel, ingress og beskrivelse. Mindre endringen ligger neders i en tabell under overskriften *Andre endringer*.



KUNDESIDER

REVISJONSNYTT CORDEL INNE 1.0

Her finner du informasjon om de vikstigste endringene/korreksjonene i Cordel Inne 1.0.

Release 2.1 - Kuler og Krutt

Onsdag 6. Mai 2020

Access Data Center features on your current Server infrastructure

Data Center was originally designed as a clustered deployment option to help enterprises improve application uptime, and maintain consistent performance at scale. Since then, we'd added more security, compliance, and administration features exclusive to Jira Data Center, many of which are available on non-clustered architecture. See the full list of features

Audit like a Pro with our revamped Audit log

Can audit logs be fun to work with? Our revamped Audit log sure is.
First of all, it's the way it looks. With the clear, user-friendly interface, you can easily find your way around it, adjust the settings, and filter the events. But as usual, it's not all about the looks. It's also what you can do with it. And there is plenty to talk about:

Andre endringer

- Kan nå bruke websiden på Internet Explorer 6, selv om dette IKKE er anbefalt
- Nettleseren koker ikke lengre over når man legger inn faktura.
- Nettsiden har nå støtte for autofyll.

Figur 4.6: Release artikkel

4.1.3 Administrator-side - Release Note System

Ved innlogging blir man omdirigert til adminsidene. Adminsidene består av to tabs, en for release note systemet og en for Azure integrasjon. Første tab er for release note systemet.

Hvordan denne siden ser ut vises i fig. 4.7. Først får man presentert en oversikten over systemet. Det er fire rader, som kan utvides. Ved utvidelse av disse får man oversikt over dataen som finnes i systemet for valgte ressurs, samt muligheter for å utføre administrative handlinger som redigering og sletting. For ytterligere informasjon se brukermanualen vedlegg H, kap. 2, AdminPanelRNS.

Figur 4.7: Admin-side

4.1.4 Administrator-side - Azure DevOps

Figur 4.8: Azure tab på admin side

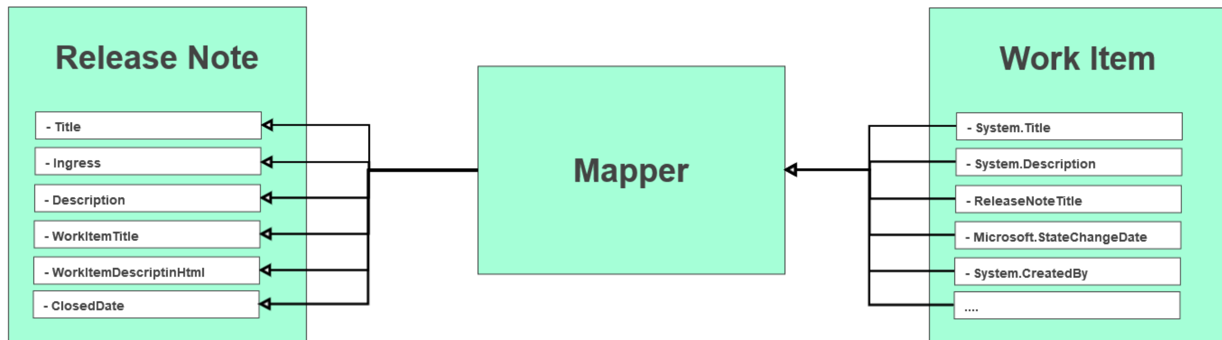
Azure DevOps tabben, vist i fig. 4.8 på administrator-siden gir to hovedfunksjoner, *Importering av releases*, og *Mapping*. Dette er tilleggsfunksjoner som forenkler prosessen å lage en release. For å kunne bruke disse funksjonene, må feltene under *Azure Pålogging* utfylles.

Importering av releases

Med importering av releases-funksjonen, er det mulig å automatisk opprette Releases og Release notes, ved å hente data fra brukerens *release pipeline* i Azure DevOps. Denne informasjonen blir brukt til å opprette en fullstendig Release med Release Notes.

Mapping

Mapping funksjonen supplerer *importering av releases* funksjonen. Her er det mulig å konfigurere felt mellom en Work Item fra Azure DevOps, og en Release Note, som brukes til å populere feltene til Release Notes når *importering av releses* brukes. En illustrasjon av dette vises i 4.9.



Figur 4.9: Sammenheng mellom Release Note og Work Item

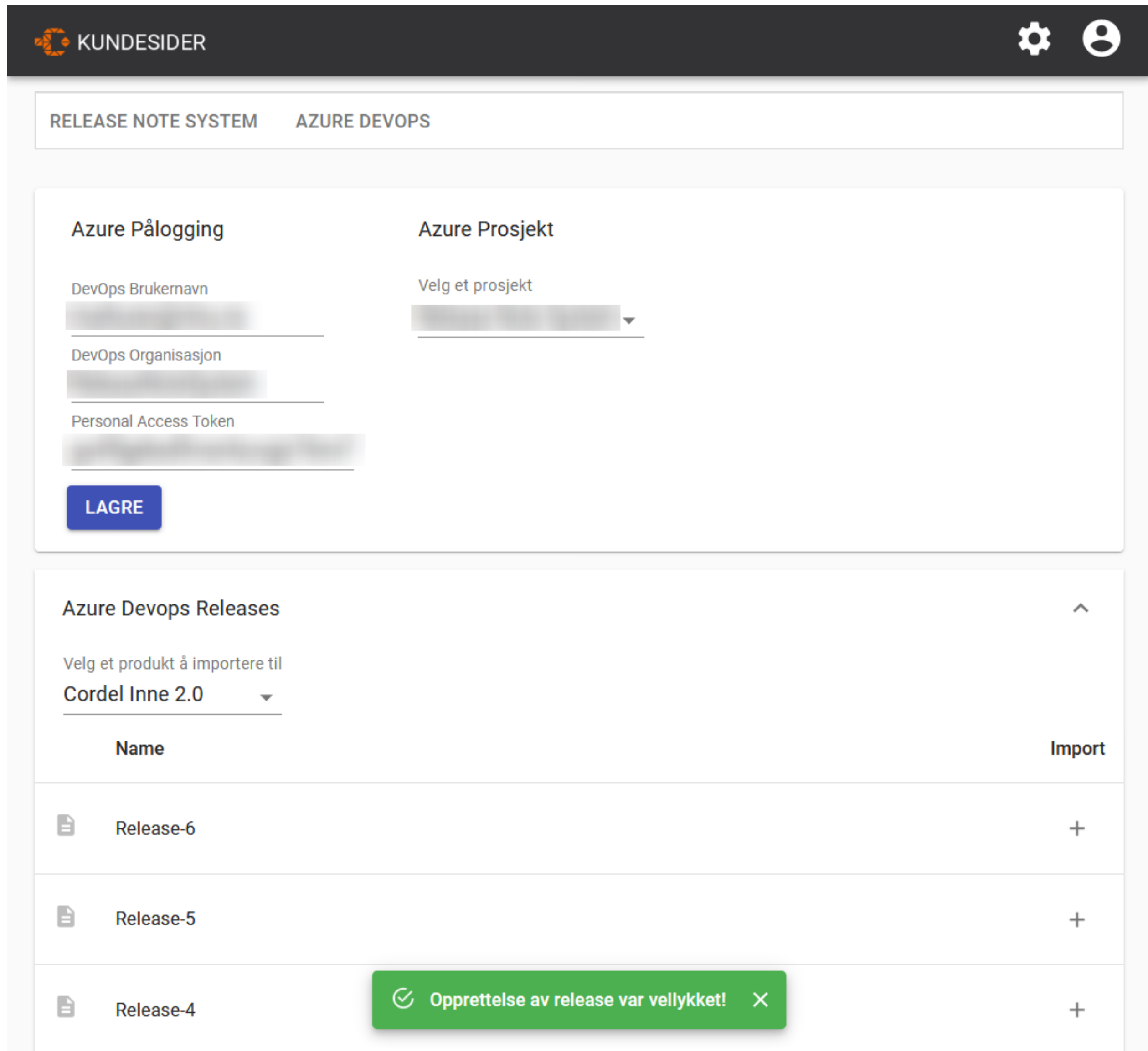
4.1.5 Login

For å komme seg til innloggingsiden må brukeren trykke en lenke som ligger i bunnen av siden. Etter en vellykket innlogging blir brukeren omdirigert til administartorsiden.

4.1.6 UI Tilbakemeldinger

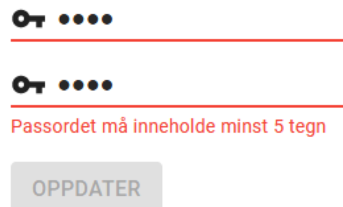
Flere administrative handlinger får tilbakemeldinger i form av notifikasjoner. Eksempel på dette vises i fig. 4.10.

En loading bar finnes i toppen av skjermbildet. Denne aktiveres når det finnes API kall som fortsatt venter på respons.



Figur 4.10: Eksempel på tilbakemelding via notifikasjon

Alle skjema i systemet har lokal validering som sjekker om diverse krav er oppfylt. For eksempel på endring av passord så stilles det krav til passord lengde.



Figur 4.11: Passord tilbakemelding

4.1.7 REST API

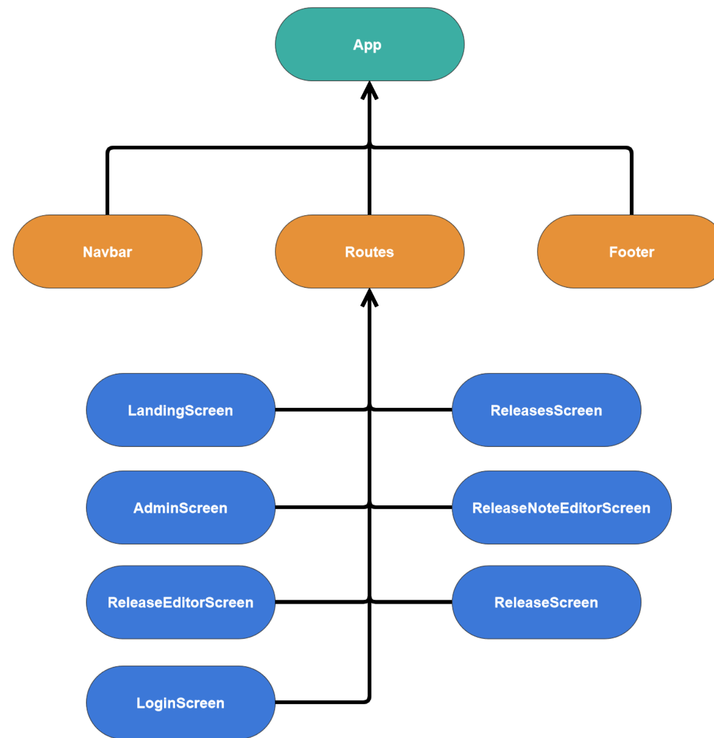
For å kommunisere mellom frontend og backend er det laget et REST API. Alle endepunkter i denne APIen er dokumentert i postman. Se vedlegg G eller på nett.

4.2 Frontend-arkitektur

Den overordnede arkitekturen i frontend er en sammensetning av React for å vise brukergrensesnittet, og Redux for å hente og lagre data fra API. Det er også brukt flere ytterlige biblioteker, som beskrevet i kap. 3.4.1. Det skal videre beskrives hvordan alle disse verktøyene er brukt i applikasjonen, og hvordan de forskjellige delene henger sammen.

4.2.1 React

Den overordnede arkitekturen av React-komponenter illustreres i 4.12. Det man kan tolke av bilde er at den øverste komponenten App fungerer som rotnode i grafen, der komponent-hierarkiet alltid består av *Navbar*, *Footer* og *Routes*.



Figur 4.12: Oversikt over komponent-arkitektur

Komponentene *Navbar* og *Footer* er enkle komponenter plasseres på toppen og bunnen av skjermbildet, mens *Routes* er komponenten som danner hovedbildet i skjermbildet. Det komponenten *Routes* viser frem vil bestemmes av *Router* 3.4.1, der alternativene består av alle komponentene i hiearkiet under *Routes*.

Routing mellom sider

For at en bruker skal kunne gå i mellom sidene på nettsiden, er React-router benyttet. Måten dette er benyttet på har allerede blitt snakket om i kap. 3.4.1. Routing er konfigurert ved å sette opp *Route* komponenter i *Routes*. Hvordan denne koden ser ut er vist i lst. 4.1.

Listing 4.1: Kode som viser routing-konfigurasjonen

```
<>
  <Route path="/" exact component={LandingScreen} />
  <Route path="/releases" exact component={ReleasesScreen} />
  <Route path="/admin/">
    <AdminScreen />
  </Route>

  <Route
    path="/releasenotes/edit/:id"
    exact
    render={(props) => <ReleaseNoteEditorScreen {...props} />}
  />
  <Route
    path="/releasenotes/create"
    exact
    component={ReleaseNoteEditorScreen}
  />
  <Route
    path="/release/:id"
    exact
    render={(props) => <ReleaseScreen {...props} />}
  />
  <Route path="/login/" exact component={LoginScreen} />
  <Route path="/releases/create" exact component={ReleaseEditorScreen} />
  <Route path="/releases/edit/:id" exact component={ReleaseEditorScreen} />
</>
```

4.2.2 Komponentdesign

I implementasjonen av komponenter til systemet, er det til beste evne fulgt et ideal om å lage gjenbrukbare komponenter. Alle komponenter er som beskrevet i kap. 3.4.1 implementert som funksjonelle komponenter. I tillegg er konseptet om *Screen* komponenter implementert for å oppnå *seperation of concerns* mellom behandling og visning av data.

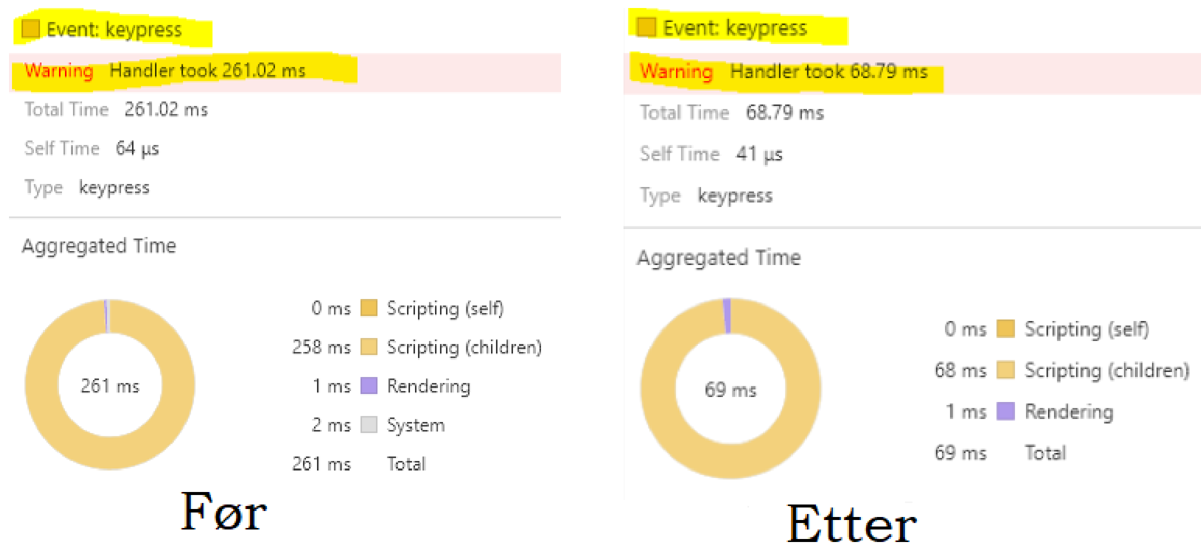
Screen komponentene, som illustrert i fig 4.12, er de øverste komponentene som utgjør en side i applikasjonen. Funksjonen til Screen komponenter er å anskaffe og levere all data som trengs av komponentene under seg. Dette

gjøres ved at *Screen* utfører alle API kall som trengs for den siden, samt å definere eventuelle callback funksjoner trengs av komponenter under. Dataen og callbacks blir gitt som props til underkomponenter. Dette designprinsippet gjør at implementasjonen av komponenter som skal utføre selve logikken på siden, blir mindre kompleks. Samtidig blir det tydeligere for utvikleren når og hvor kommunikasjon med nettverket skjer.

4.2.3 Optimalisering av Ytelse

Metodene for optimalisering av react komponenter som beskrevet i kap. 3.4.1, er tatt i bruk å øke ytelse i *ReleaseEditor*. Ytelsen i denne komponenten nådde et punkt der det gikk ut over praktisk bruk. Det ble så med Reacts Profiler identifisert at gjengivelse av lange lister med komponenter var kilden til den dårlige ytelsen, og implementert memoization på disse komponentene.

Figur 4.13 viser måling av ytelse før og etter implementert optimisering. Det som er målt her er tiden til en gjengivelse som er trigget av å skrive i en tekst-input. Problemet her er altså at for hvert tastetrykk i tekst-inputen, måtte brukeren vente omtrent 260ms før brukergrensesnittet ble oppdatert. Etter optimalisering, er denne tiden redusert til omtrent 70ms.



Figur 4.13: FØR og ETTER optimalisering av ytelse

4.2.4 Bruk Av Axios

Axios er som forklart i kap. 3.4.1 brukt for å utføre nettverkskall. Her skal det vises hvordan disse funksjonene er implementert i systemet.

I de alle fleste tileller er det brukt en global instans av axios, med en default *base url* (lst. 4.2), en interceptor på utgående requests for å sette JWT token for autorisering (lst. 4.3), samt en interceptor på inkommende responses for å redirecte til login om statuskoden er 401 (lst. 4.4). Generell bruk av axios blir da som illustrert i lst. 4.5. Det er altså ikke nødvendig å skrive inn full URL eller headers for å gjøre et nettverkskall.

Listing 4.2: Axios default base url

```
1 Axios.defaults.baseURL = process.env.REACT_APP_APP_URL + "/api/";
```

Listing 4.3: Axios interceptor på utgående request

```
1 Axios.interceptors.request.use((request) => {
2   const token = getAuthToken();
3   if (token) {
4     request.headers.Authorization = `Bearer ${token}`;
5   }
6   return request;
7 });
```

Listing 4.4: Axios interceptor på inkommende request

```
1 Axios.interceptors.response.use(
2     (response) => response,
3     (error) => {
4       if (error.response?.status === 401) {
5         history.push("/login");
6       }
7       return Promise.reject(error);
8     }
9   );
```

Listing 4.5: Generell bruk av axios

```
Axios.get("products/")
  .then((res) => {
      .
      .
      .
  })
  .catch((error) => {
      .
      .
      .
  });
```

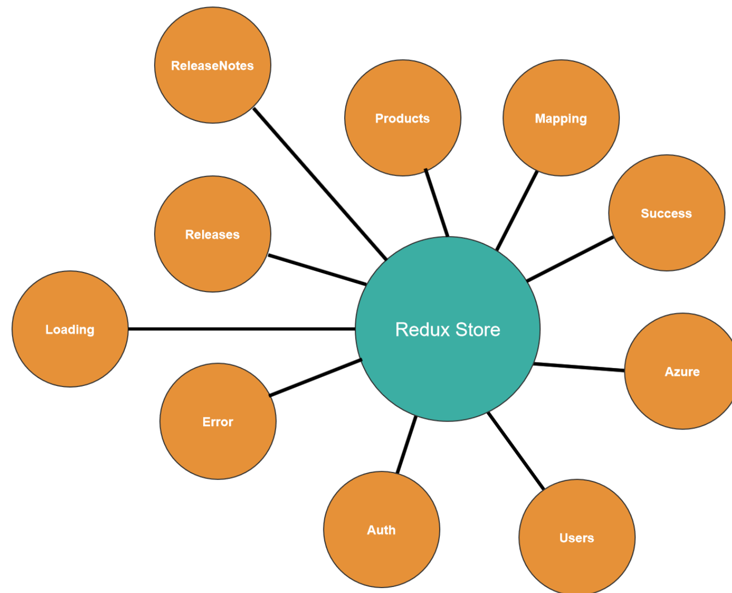
Ett tilfelle som krever ulik konfigurasjon fra den globale Axios instansen, er når det gjøres kall til Azure sitt API. Her er det skapt en lokal instans, som er separat fra konfigurasjonen til den globale instansen (lst. 4.6).

Listing 4.6: Creating local instance of Axios

```
export const AzureAxios = GlobalAxios.create({
  timeout: 5000,
});
```

4.2.5 Tilstandsbehandling

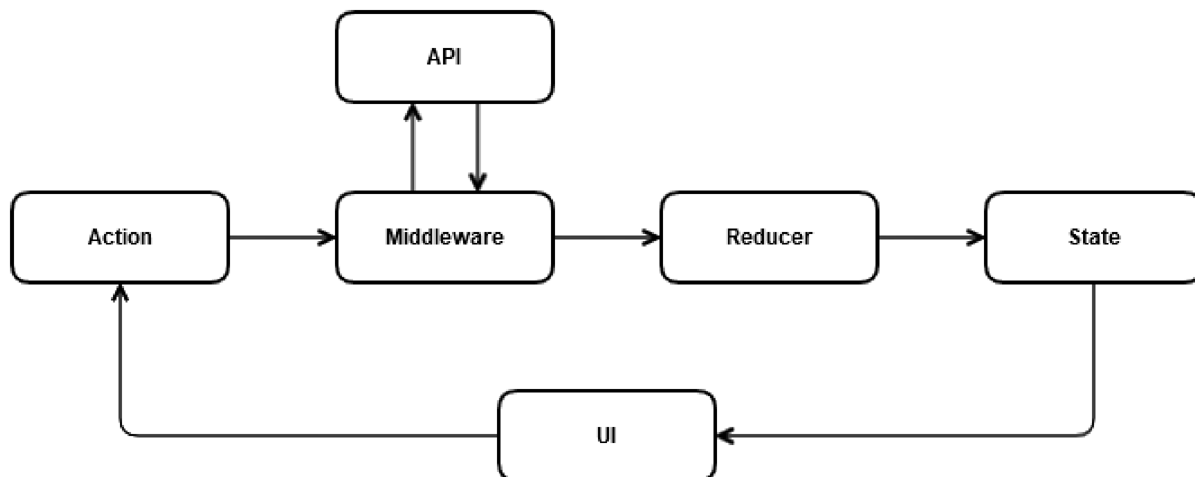
Den globale tilstanden er illustrert i form av en graf i fig. 4.14. Det man kan tolke av bildet er at det eksisterer flere separate reducer-tilstander som danner den globale tilstanden. Her baserer skille seg på hvilke entiteter som eksisterer i systemet.



Figur 4.14: Graf som viser den globale tilstanden.

Her skal det først vises hvordan metodene beskrevet i kap. 3.4.1 er implementert, og deretter hvordan Redux tilstanden leses og manipuleres fra React komponenter.

I kap. 3.4.1 er det beskrevet hvordan Redux brukes for å ha en *single source of truth* for data fra API-en, og at det skal være et tydelig skille mellom logikken for å hente og lagre, og for å vise data. De forskjellige delene som brukes for å oppnå dette er illustrert i fig . Videre skal det beskrives hvordan disse delene er implementert.



Figur 4.15: Informasjonsflyt med redux.

Actions

Det brukes tre tilstander for å definere tilstanden til et request; Pending, Error, og Success. Kodesnutten 4.7 viser typisk opprettelse av tre *action creators*, som er funksjoner som kan kalles for å returnere et action objekt. Denne opprettelsen skjer ved hjelp av `createAction(...)` funksjonen fra *redux-toolkit*.

Listing 4.7: Actions Eksempel

```
1 export const fetchProductsPending = createAction("products/getPending");
2 export const fetchProductsError = createAction("products/getError");
3 export const fetchProductsSuccess  = createAction("products/getSuccess");
```

Thunks

Lst 4.8 viser implementasjonen av ett typisk API-kall gjennom redux, som blir referrert til som en *thunk*. Det som skjer i dette eksempelet er at det returneres en funksjon til redux sine middleware komponenter, der *redux-thunk* er installert, og sørger for å fullføre nettverkskallet, og asynkron dispatching av actions. Det blir altså først sendt at et request er underveis. Deretter blir det sendt at requesten enten er vellykket eller feilet, med dataen fra responen.

Listing 4.8: Thunk eksempel

```
1  function fetchProducts() {
2    return (dispatch) => {
3      dispatch(fetchProductsPending());
4      return Axios.get("products/")
5        .then((res) => {
6          dispatch(fetchProductsSuccess({ data: res.data }));
7        })
8        .catch((error) => {
9          dispatch(fetchProductsError(error));
10       });
11   };
12 }}
```

Reducers

Reducers mottar actions og bestemmer derfra hvordan tilstanden skal oppdateres. Det er implementert to hovedgrupper av reducers; Reducers som bryr som om å lagre data fra requests, og reducers som håndterer brukersentrert informasjon, som respons-meldinger og loading flagg.

Reducers som bryr seg som om å lagre data tar for seg å lagre data fra requests og er implementert som illustrert i fig. 4.9. Her kan man se at en reducer er opprettet ved å bruke `createReducer()` funksjonen fra *Redux-Toolkit*. Den-

ne funksjonen oppretter en reducer med to argument. Først oppgis den initelle tilstanden til reduceren. Deretter oppgis alle *actions* from skal behandles av reduceren, samt logikken for hvordan data fra gitte *action* skal lagres.

Listing 4.9: Reducer eksempel

```
const productsReducer = createReducer(
  { items: [] },
  {
    [fetchProductsSuccess]: (state, action) => {
      state.items = action.payload.data;
    },
    [putProductVersionSuccess]: (state, action) => {
      const product = state.items.find((p) => p.id === action.payload.id);
      product.versions.push(action.payload.data);
    },
  }
);
```

Reducers som håndterer brukersentrert informasjon, skiller seg ved at de ikke tar for seg spesifikke actions. I stedet bruker de Regex til å se på den generelle typen action. Dette går ut fra at actions følger en fast navnekonvensjon.

Listing 4.10: loadingReducer eksempel

```
export const loadingReducer = (state = {}, action) => {
  const { type } = action;
  const matches = /(.*)(Pending|Success|Error)/.exec(type);

  // not a *PENDING / *SUCCESS / *ERROR actions, so we ignore them
  if (!matches) return state;

  const [, requestName, requestState] = matches;
  return {
    ...state,
    // Store whether a request is happening at the moment or not
    // e.g. will be true when receiving "products/getPending"
    // and false when receiving "products/getSuccess" or "products/getError"
    [requestName]: requestState === "Pending",
  };
};
```

Reduceren i lst. 4.10 viser implementasjonen `loadingReducer`, som lagrer på loading flagg for hver request ved å finne ut om action typen inkluderer enten *Pending*, *Success*, eller *Error* i navnet. Ekvivalente reducere er også implementert for Error og Success-meldinger.

Hvordan Redux-tilstanden leses og manipuleres fra React komponenter

Redux gir tilgang til to hooks som brukes for å lese og manipulere tilstanden, `useDispatch` og `useSelector`.

For å sende informasjon til redux er `useDispatch` brukt. Denne hooken gir tilgang til dispatch-funksjonen i redux. Lst. 4.11 viser eksempel på typisk bruk. Først anskaffes dispatch funksjonen. Deretter blir dispatch-funksjonen kalt med en *thunk*. Dette skjer også inne i en `useEffect` hook. Det denne gjør er å sørge for at funksjonen kun blir kalt ved først innlasting av komponenten.

Listing 4.11: useDispatch eksempel

```
1   const dispatch = useDispatch();
2
3   useEffect(() => {
4     dispatch(fetchProducts());
5   }, [dispatch]);
```

For så å lese data fra tilstanden, brukes `useSelector`. Denne hooken tar en funksjon som argument, som sier hvilke data som skal hentes ut. Eksempel på dette er gitt i lst. 4.12. Dataen som hentes fra useSelector oppdateres automatisk når redux tilstanden oppdateres.

Listing 4.12: useSelector eksempel

```
1   const products = useSelector((state) => state.products);
```

4.2.6 Testing

Tester som er implementert i frontend er tester for alle thunks og reducers, samt noen tester for React komponenter. Disse testene er som beskrevet i kap. 3.4.1 implementert med *Jest* og *React-testing-library*.

Ved testing av thunks er det tatt i bruk conformance testing, som beskrevet i 3.7.4. Conformance testing ble implementert ved å lage en funksjon som kalles `testThunkConformance`. Denne funksjonen kalles med ønsket *thunk* funksjon som argument. Det blir så testet om gitte *thunk* funksjonen produserer alle forventede resultat.

Azure pipelines 3.7.1 brukes for å automatisk kjøre disse testene ved pull requests på GitHub. Stegene som er implementert i `azure-pipelines.yml` er som følger:

- Installer Node.js
- Let etter cachet verjon av package-lock.json
- Installer dependencies
- Kjør tester

- Publiser test-resultat og coverage rapporter.

Etter disse stegene har kjørt, blir også resultatet av kjøringen publisert til pull requesten i GitHub som trigget kjøringen. Alle test-resultat og coverage rapporter kan finnes i prosjektets Azure DevOps Pipelines. Det er også lagt ved eksempel på en coverage rapport, se vedlegg F.

4.2.7 Visning av releases

Et krav i visningen av releases, var det skal finnes et skille mellom større og mindre endringer; Større endringer skal vises øverst, med full *Title*, *Ingress*, og *Description*. Mindre endringer skal vises nederst i et eget avsnitt, "Andre endringer". Siden en release note i seg selv ikke inneholder meta-informasjon som sier hvilken kategori den tilhører, er det implementert egen logikk for å bestemme dette i front-end. I kildekoden er denne klassifiseringen beskrevet som *FULL* og *DENSE* Release Notes. En resulterende Release-side er vist i fig. 4.5.

Logikk-tabellen i fig 4.2.7 viser hvordan dette er implementert. Altså, alle Release Notes som mangler tittel blir klassifisert som *DENSE*, og går under andre endringer, mens resten blir klassifisert som *FULL*.

T = Title

I = Ingress

D = Description

C = Classification

T	I	D	C
1	0	0	*FULL*
1	1	0	*FULL*
1	1	1	*FULL*
1	0	1	*FULL*
0	0	1	*DENSE*
0	1	1	*DENSE*
0	0	0	*DENSE*

4.3 Backend-arkitektur

I dette kapitellet gjennomgås det hvordan ASP.NET Core er konfigurert. Det som blir gjennomgått her er viktige klasser og deres funksjoner, samt hvordan testing er gjennomført.

4.3.1 ASP.NET Core

Implementasjonen av REST API i ASP.NET Core er gjort ved å bruke tre lag for å håndtere forespørsler, business logic og snakke med database. Disse tre lagene blir henholdvis kalt for Controller, service, og repository.

Startup

Det som blir gjennomgått nå er en oversikt over gjennomført konfigurering av services, autentiseringsløsning, endepunkter. Services er implementert som beskrevet i kap. 3.4.2.

Oversikt over hver enkelt service av klassetype repository og service kan sees i figur 4.13.

Listing 4.13: Oversikt over alle repository- og serviceklassene

```
// BIND ALL REPOS
services.AddScoped<IUserRepository, UserRepository>();
services.AddScoped<IArticleRepository, ArticleRepository>();
services.AddScoped<IProductRepository, ProductRepository>();
services.AddScoped<IReleaseNoteRepository, ReleaseNoteRepository>();
services.AddScoped<IReleaseRepository, ReleaseRepository>();
services.AddScoped<IProductVersionRepository, ProductVersionRepository>();
services.AddScoped<IAzureInformationRepository, AzureInformationRepository>();
services.AddScoped<IMappableRepository, MappableRepository>();
services.AddScoped<IImageRepository, ImageRepository>();

// BIND ALL SERVICES
services.AddScoped<IUnitOfWork, UnitOfWork>();
services.AddScoped<IAuthenticationService, AuthenticationService>();
services.AddScoped<IArticleService, ArticleService>();
services.AddScoped<IUserService, UserService>();
services.AddScoped<IProductService, ProductService>();
services.AddScoped<IReleaseNoteService, ReleaseNoteService>();
services.AddScoped<IReleaseService, ReleaseService>();
services.AddScoped<IProductVersionService, ProductVersionService>();
services.AddScoped<IImageService, ImageService>();
services.AddScoped<IMappableService, MappableService>();
```

Konfigurasjonen til autentiseringsløsningen er også konfigurert i denne klassen. Oversikten over denne finnes i lst. 4.14.

Listing 4.14: Konfigurasjonen til autentiseringsløsningen. Det man kan se her hvordan JWT ble konfigurert og konfigurasjonen til signering.

```
1  // CONFIGURE AUTHENTICATION AND TOKEN OPTIONS
2  services.Configure<TokenOptions>(Configuration.GetSection("TokenOptions"));
3  var tokenOptions = Configuration.GetSection("TokenOptions")
4              .Get<TokenOptions>();
5
6  var signingConfigurations = new SigningConfigurations();
7  services.AddSingleton(signingConfigurations);
8
9  services.AddAuthentication(x =>
10     {
11         x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
12         x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
13     })
14             .AddJwtBearer(jwtBearerOptions =>
15     {
16         jwtBearerOptions.RequireHttpsMetadata = false;
17         jwtBearerOptions.SaveToken = true;
18         jwtBearerOptions.TokenValidationParameters = new TokenValidationParameters()
19         {
20             ValidateAudience = true,
21             ValidateLifetime = true,
22             ValidateIssuerSigningKey = true,
23             ValidIssuer = tokenOptions.Issuer,
24             ValidAudience = tokenOptions.Audience,
25             IssuerSigningKey = signingConfigurations.Key,
26             ClockSkew = TimeSpan.Zero
27         };
28     });
```

Oppstarts-konfigurasjonen til databaseløsningen er konfigurert som vist i lst. 4.15.

Listing 4.15: Konfigurasjonen til databasen. Her ser man at miljøvariablene hentes ut og brukes til å opprette tilkoblingsstrengen til databasen og at det legges inn en database-service.

```
1  // GRABBING ENVIRONMENT VARIABLES
2  // Need to do the construction of the connectionString inside StartUp,
3  // since it's hard to do string manipulation in docker-world
4  var host = Environment.GetEnvironmentVariable("HOST");
5  var port = Environment.GetEnvironmentVariable("PORT");
6  var db = Environment.GetEnvironmentVariable("DB_NAME");
7  var user = Environment.GetEnvironmentVariable("DB_USER");
8  var passw = Environment.GetEnvironmentVariable("DB_PASSW");
9  var connectionString =
10   "host=" + host + ";port=" + port + ";database=" + db + ";username=" + user + ";
       password=" + passw + ";";
11
12 services.AddHttpContextAccessor();
13 // ADDS DATABASE SERVICE
14 // CONNECTS WEB-API TO DB
15 services.AddDbContext<AppDbContext>(options => { options.UseNpgsql(connectionString);
16 });
```

Konfigurasjonen for middleware kan man se i lst. 4.16

Listing 4.16: Konfigursjon av middleware. Det man ser i dette bildet er hvilken middleware som ble brukt i systemet, der de mest essensielle er autentisering, autorisering og endepunkt. Det ble også lagt til rette for å legge til HTTPS i fremtiden

```
1  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
2  {
3      if (env.IsDevelopment())
4      {
5          IdentityModelEventSource.ShowPII = true;
6          app.UseDeveloperExceptionPage();
7      }
8      else
9      {
10         app.UseHsts();
11     }
12
13     // Init static file serving
14     var webRoot = Directory.CreateDirectory(Path.Combine(Directory.GetCurrentDirectory()
           , "wwwroot"));
15     env.WebRootPath = webRoot.FullName;
16     Directory.CreateDirectory(Path.Combine(env.WebRootPath, "images"));
17     app.UseStaticFiles();
18
19     // SHOULD BE ENABLED IN PRODUCTION ENVIRONMENTS
20     // app.UseHttpsRedirections();
21
22     app.UseRouting();
23     app.UseCors(MyAllowSpecificOrigins);
24     app.UseAuthentication();
25     app.UseAuthorization();
26     app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
27 }
```

Controller-klasse

En Controller er definert for nesten hver Modell i systemet med metoden som ble nevnt i kap. 3.4.2. De entitetene som ikke har eget endepunkt, har relasjoner til andre Modeller og blir aksesserbare gjennom disse. Den fullstendige oversikten over Controllers som finnes i systemet, ligger i vedlegg N.

Service-klasse

Det finnes en Service-klasse for hver Controller, der hver Service implementerer et interface som definerer alle oppgavene til Servicen. Den fullstendige oversikten over Service-klasser finnes i vedlegg P. Et eksempel på en hente-funksjon i en Service-klasse finnes i lst. 4.17.

Listing 4.17: Utsnitt fra ReleaseService.cs som henter data fra et repository, og deretter returnerer en respons.

```
1  public async Task<ReleaseResponse> GetRelease(int id)
2  {
3      var existingRelease = await _releaseRepository.FindByIdAsync(id);
4      if (existingRelease == null)
5      {
6      return new ReleaseResponse("Releasen eksisterer ikke!");
7      }
8
9      try
10     {
11         return new ReleaseResponse(existingRelease);
12     }
13     catch (Exception e)
14     {
15         return new ReleaseResponse($"Det oppsto en feil: {e.Message}");
16     }
17 }
```

Repository-klasse

Det finnes en Repository-klasse som oppfyller krav til implementasjon for sine respektive interface for hver Modell. Den fullstendige oversikten over
Repository-klasser finnes i vedlegg P. Et eksempel på hvilke metoder som befinner seg i en Repository-klasse finnes i lst. 4.18.

Listing 4.18: Utsnitt fra ReleaseRepository.cs som viser metoder for å legge til-, slette-, oppdatere- og hente data.

```
public async Task AddAsync(Release release)
{
    await _context.Releases.AddAsync(release);
}

public void Remove(Release release)
{
    _context.Releases.Remove(release);
}

public void Update(Release release)
{
    _context.Releases.Update(release);
}

public async Task<Release> FindByIdAsync(int id)
{
    return await _context.Releases
        .Include(r => r.ProductVersion)
        .ThenInclude(pv => pv.Product)
        .Include(r => r.ReleaseReleaseNotes)
        .ThenInclude(rrn => rrn.ReleaseNote)
        .SingleOrDefaultAsync(r => r.Id == id);
}
```

4.3.2 Test resultater

I denne seksjonen gjennomgås det ulike utfall test-containerer kan få, test feilet og ingen test feiltet.

4.3.3 Test har feilet

I dette tilfellet har det skjedd noe galt med testene, da kan man se på figur 4.16 at 5 tester passerte stilte krav, mens eksekveringen har stoppet på en av testene. I dette bilde kan man se at årsaken til at testen feilet er på grunn av "*Uncaught AssertionError: expected undefined to be a number*", dette betyr at man har anledning til å starte

feilsøkingen med denne informasjonen som utgangspunkt.

```
test-runner_1      | 2020/05/09 14:08:57 Problem with dial: dial tcp 172.19.0.3:5000: getsockopt: connec
tion refused. Sleeping 1s
test-runner_1      | 2020/05/09 14:08:58 Connected to tcp://dockerapi:5000
test-runner_1      | yarn run v1.22.4
test-runner_1      | warning package.json: No license field
test-runner_1      | $ mocha --timeout 10000 --bail --file test/testAuth.js
test-runner_1      |
test-runner_1      |
test-runner_1      |   Login
test-runner_1      |     ✓ Should return bad request (491ms)
test-runner_1      |     ✓ Should return token (120ms)
test-runner_1      |
test-runner_1      |   Mappings GET
test-runner_1      |     ✓ should return unauthorized
test-runner_1      |     ✓ should return unauthorized
test-runner_1      |     ✓ should return all mappable fields (121ms)
test-runner_1      |     1) should return mapped fields
test-runner_1      |
test-runner_1      |   5 passing (796ms)
test-runner_1      |   1 failing
test-runner_1      |
test-runner_1      |   1) Mappings GET
test-runner_1      |        should return mapped fields:
test-runner_1      |      Uncaught AssertionError: expected undefined to be a number
test-runner_1      |       at /app/test/testMappings.js:75:45
test-runner_1      |       at Test.Request.callback (node_modules/superagent/lib/node/index.js:716:12)
test-runner_1      |       at /app/node_modules/superagent/lib/node/index.js:916:18
test-runner_1      |       at IncomingMessage.<anonymous> (node_modules/superagent/lib/node/parsers/json
.js:19:7)
test-runner_1      |       at endReadableNT (_stream_readable.js:1223:12)
test-runner_1      |       at processTicksAndRejections (internal/process/task_queues.js:84:21)
test-runner_1      |
test-runner_1      |
test-runner_1      |
test-runner_1      |
test-runner_1      | error Command failed with exit code 1.
test-runner_1      | info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command
.
test-runner_1      | 2020/05/09 14:08:59 Command exited with error: exit status 1
releasenotes-webapi_test-runner_1 exited with code 1
```

Figur 4.16: Test-rapport på test har feilet.

4.3.4 Test er vellykket

I dette tilfellet har alle testene gått igjennom og man kan være sikker på at alt som testes fungerer som de skal.

```
test-runner_1      |   Release notes DELETE
test-runner_1      |     ✓ DELETE | Tries to delete a release note without being logged in
test-runner_1      |     ✓ DELETE | Tries to delete a non-existant release note
test-runner_1      |     ✓ DELETE | Should return a OK 200 and a copy of the deleted release note
test-runner_1      |     ✓ DELETE | Tries to delete a already deleted release note
test-runner_1      |
test-runner_1      |   Releases POST
test-runner_1      |     ✓ CREATE | Tries to create a release without being logged in
test-runner_1      |     ✓ PUT | Tries to update a release without being logged in
test-runner_1      |     ✓ CREATE | Successfully creating a release (103ms)
test-runner_1      |     ✓ CREATE | Successfully creating a release with non existing Release Note
test-runner_1      |     ✓ CREATE | Tries to create a already created release
test-runner_1      |     ✓ POST | Should create release with raw workItems (49ms)
test-runner_1      |     ✓ PUT | Should return same object as trying to put  (52ms)
test-runner_1      |     ✓ PUT | Should return non null fields
test-runner_1      |     ✓ PUT | Should handle unknown values
test-runner_1      |
test-runner_1      |   Releases GET
test-runner_1      |     ✓ Should return all releases (53ms)
test-runner_1      |     ✓ GET | Attempting to get a non-existant release
test-runner_1      |     ✓ GET | Should return a specific release
test-runner_1      |     ✓ GET | Only retrieve Releases that's public
test-runner_1      |
test-runner_1      |   Releases DELETE
test-runner_1      |     ✓ DELETE | Tries to delete a release without logging in
test-runner_1      |     ✓ DELETE | Tries to delete a non-existant release
test-runner_1      |     ✓ DELETE | Should delete a release and return a copy of the deleted release
test-runner_1      |     ✓ DELETE | Tries to delete a already deleted release
test-runner_1      |
test-runner_1      |   Users POST
test-runner_1      |     ✓ POST | Should return unauthorized
test-runner_1      |     ✓ POST | Should return created
test-runner_1      |     ✓ POST | Should return email already in use
test-runner_1      |
test-runner_1      |   Change password
test-runner_1      |     ✓ PUT | Password got changed successfully
test-runner_1      |     ✓ POST | Logging in with new password
test-runner_1      |     ✓ PUT | The user doesn't exist
test-runner_1      |     ✓ PUT | Can't use the same password
test-runner_1      |     ✓ PUT | Atleast 5 characters in new password
test-runner_1      |
test-runner_1      |   Azure Info
test-runner_1      |     ✓ should return info
test-runner_1      |     ✓ should return unauthorized
test-runner_1      |
test-runner_1      |
test-runner_1      |   47 passing (2s)
test-runner_1      |
test-runner_1      | Done in 2.00s.
test-runner_1      | 2020/05/09 16:20:12 Command finished successfully.
releasenotes-webapi_test-runner_1 exited with code 0
```

Figur 4.17: Test-rapport på test er vellykket.

4.3.5 Oppbygging

Måten hver enkelt test ble kjørt på var gjennom bruk av hovedsaklig tre komponenter; Node.Js, Mocha og Chai.

- NodeJs fungerte som runtime-environmentet som kjørte alle testene.
- Mocha ble brukt som test-rammeverk for å strukturere alle testene.
- Chai ble brukt for å sende HTTP-forespørsler og til assertions.

4.4 Database

Det som skal oppsummeres i dette underkapittelet er hvilke tabeller som finnes i databasen, hvilke forhold som er definert og den konfigurasjonen av dette.

4.4.1 Konfigurasjon i ASP.NET Core

Måten databasen er satt opp på med tanke på entiteter og relasjoner ble forklart i kap. 3.4.2. Som nevnt i kap. 3.4.2 blir databasen konfigurert i klassen AppDbContext. Lst. viser 4.19 hvordan hver tabell i databasen er satt opp som et felt i denne klassen.

Listing 4.19: Oversikt over alle definerte tabeller i klassen AppDbContext.

```
1     public DbSet<User> Users { get; set; }
2     public DbSet<Role> Roles { get; set; }
3     public DbSet<Release> Releases { get; set; }
4     public DbSet<ReleaseNote> ReleaseNotes { get; set; }
5     public DbSet<ProductVersion> ProductVersions { get; set; }
6     public DbSet<Product> Products { get; set; }
7     public DbSet<ReleaseReleaseNote> ReleaseReleaseNotes { get; set; }
8     public DbSet<AzureInformation> AzureInformations { get; set; }
9     public DbSet<MappableField> MappableFields { get; set; }
10    public DbSet<ReleaseNoteMapping> ReleaseNoteMappings { get; set; }
11    public DbSet<MappableType> MappableTypes { get; set; }
```

Videre blir nøkler, constraints og default-verdier konfigurert i funksjonen `OnModelCreating`. Lst. 4.4.1 viser resultatet av denne konfigurasjonen. Her kan man se at det er gjort spesifikk konfigurasjon for enkelte tabeller, der keys, default-verdi, constraints og sikkerhet blir lagt til.

- **Default-verdi** Her blir det satt default-verdi for dato.
- **Generert primary-key** Her blir lagt til generering av primary-key.
- **Constraints** Unique constraint.
- **Key Type** Her blir Composite keys konfigurert.
- **Sikkerhet** Her blir data-autorisering satt opp om beskrevet i kap 3.6.2.

```
1  protected override void OnModelCreating(ModelBuilder builder)
2  {
3      base.OnModelCreating(builder);
4
5      // Generate default values
6      builder.Entity<Release>().Property(r => r.Date).HasDefaultValue(DateTime.UtcNow);
7
8      // Generate primary key on add
9      builder.Entity<AzureInformation>().Property(ai => ai.Id).ValueGeneratedOnAdd();
               builder.Entity<ReleaseNote>().Property(rn => rn.Id).ValueGeneratedOnAdd()
           ;
10     builder.Entity<ProductVersion>().Property(pv => pv.Id).ValueGeneratedOnAdd();
11     builder.Entity<MappableField>().Property(mf => mf.Id).ValueGeneratedOnAdd();
12     builder.Entity<MappableType>().Property(mt => mt.Id).ValueGeneratedOnAdd();
13
14     // Generate unique constraint
15     builder.Entity<MappableType>().HasIndex(mt => mt.Name).IsUnique();
16
17     // Generate key types
18     builder.Entity<UserRole>().HasKey(ur => new {ur.UserId, ur.RoleId});
19     builder.Entity<ReleaseNoteMapping>()
20         .HasKey(rnm => new {rnm.MappableFieldId, rnm.MappableTypeId});
21     builder.Entity<ReleaseReleaseNote>().HasKey(
22         rrn => new {rrn.ReleaseId, rrn.ReleaseNoteId});
23
24     // All entities that require extra security for isPublic
25     builder.Entity<Release>().HasQueryFilter(
26         r => !(!r.IsPublic && !UserIsAdmin));
27     builder.Entity<ProductVersion>().HasQueryFilter(
28         pr => !(!pr.IsPublic && !UserIsAdmin));
29     builder.Entity<ReleaseNote>().HasQueryFilter(
30         rn => !(!rn.IsPublic && !UserIsAdmin));
31 }
```

4.4.2 Tabeller og Relasjoner

Forrige seksjon beskrev hvordan entitetene i databasen er konfigurert i ASP.NET Core. Denne seksjonen skal vise de resulterende entitetene og relasjonene, samt hvordan disse brukes. Databasen består av tre systemer: ReleaseNotes, Mapping, og Auth. Den fullstendige oversikten over alle tabeller og relasjoner finnes i vedlegg M.

ReleaseNotes

Disse tabellene holder på dataen som utgjør kjernefunksjonaliteten i systemet. Entitetene har her feltet "isPublic". Dette feltet brukes til å styre synligheten til entiteten, ved hjelp av data-autorisering 3.6.2. Oversikt som bare viser sammenheng mellom Release Note, Release, Product og ProductVersion kan ses i figur 4.18.

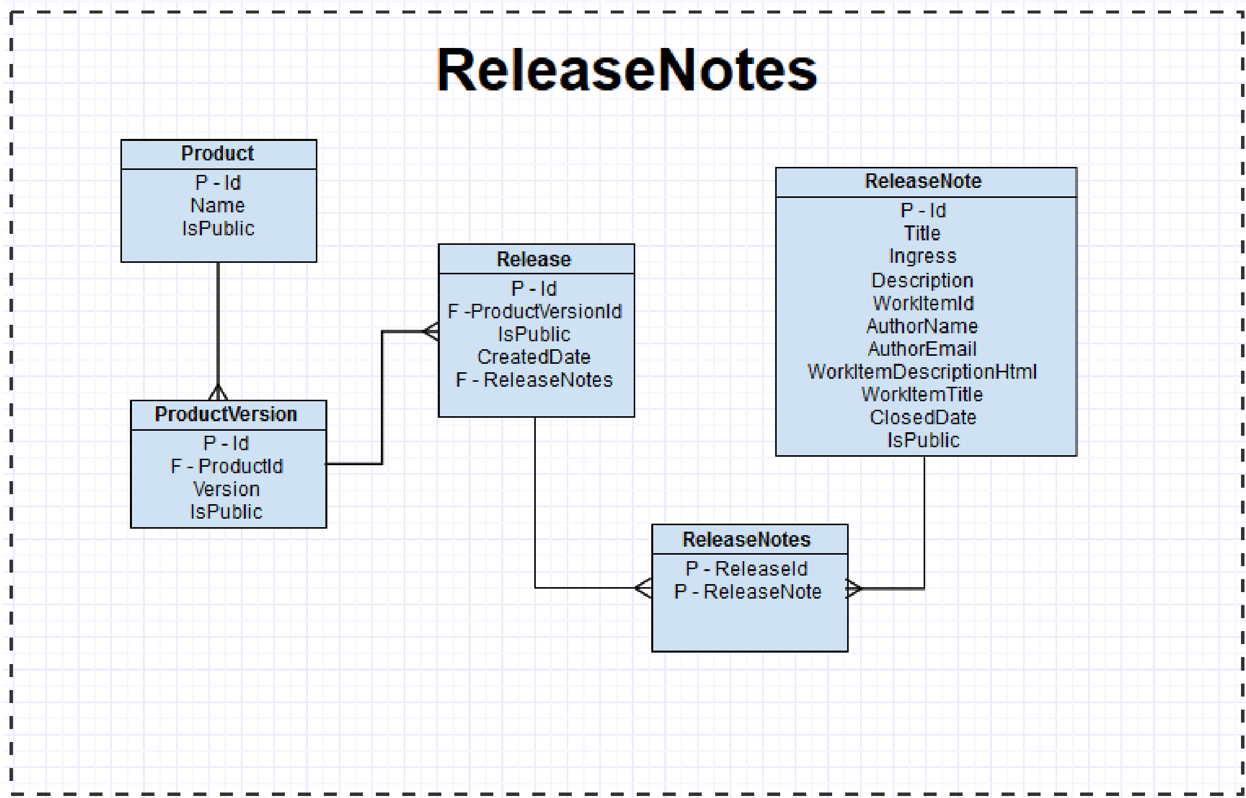
Release Note beskriver en enkelt release note. Title, ingress, og description utgjør feltene som en bruker kan redigere. WorkItemId, AuthorName, AuthorEmail, WorkItemdescription, WorkItemTitle er alle felter som som holder informasjon om opphavet til release noten.

Product Beskriver et produkt med navn og flagg som beskriver om den er offentligjort.

ProductVersion Beskriver many-to-many-forholdet mellom Release og Product. Der en Release kan tilhøre flere ProductVersions, og Product som kan ha flere versjoner.

ReleaseReleaseNotes bekriver many-to-many-forholdet mellom ReleaseNote og Release. Der en Release kan ha flere ReleaseNotes, og Release Note som kan tilhøre flere Releases.

Release Beskriver en enkelt Release. Denne består av Release Notes, ProductVersions, opprettelsesdato og flagg som beskriver om den er offentligjort.



Figur 4.18: ER-Diagram for Release Notes

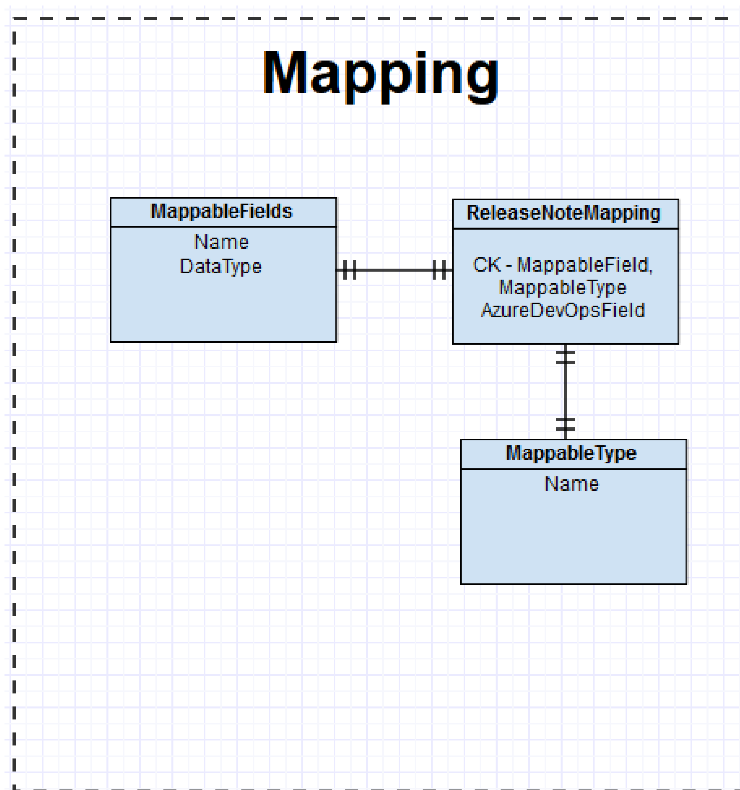
Mapping

Mapping tabellene inneholder all data som brukes av mapping-endpoints. Oversikt som bare viser sammenheng mellom MappableFields, MappableType og ReleaseNoteMapping kan ses i figur 4.19.

Mappable fields inneholder alle felt fra en ReleaseNote som er mulig å mappe.

MappableType eksisterer for å skille mellom typen work item det mappes fra. MappableTypes består av task eller bug.

ReleaseNoteMapping beskriver en fullstendig mapping. Denne entiteten består av en komposittnøkkel av MappableField og MappableType - som utgjør ene siden av mappingen, samt en AzureDevopsField som utgjør andre siden av mappingen.



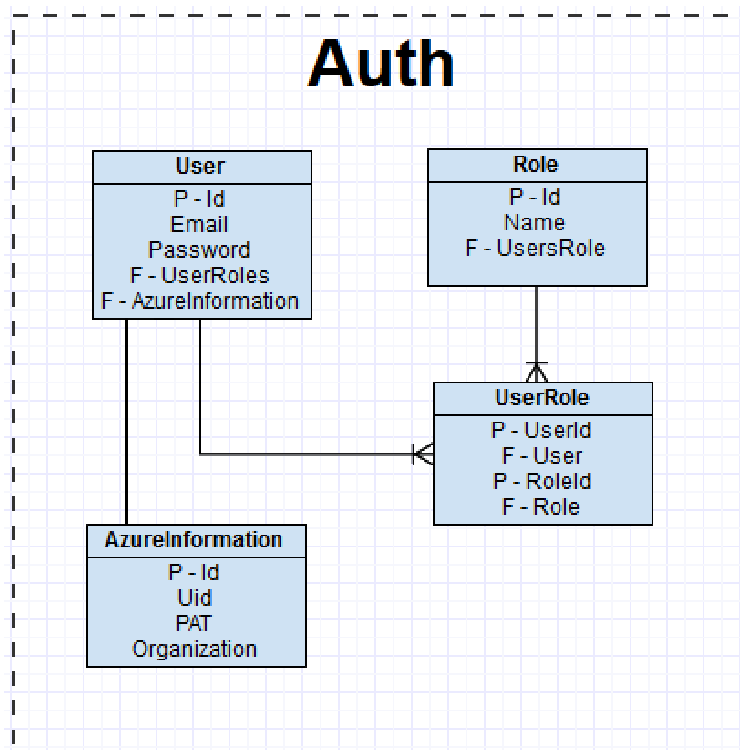
Figur 4.19: ER-Diagram over Mapping

Auth

Auth tabellene inneholder all informasjon om brukere. Oversikt som bare viser sammenheng mellom Users og AzureInformation kan sees i figur 4.20.

Hver *User*, eller bruker har i tillegg til Email og Password, en UserRole som angir rollen til brukeren, som brukes til å bestemme hvilke rettigheter en bruker skal ha i systemet.

Videre har brukeren en *AzureInformation* tabell, som inneholder informasjon som brukes til å autentisere med Azure.



Figur 4.20: ER-Diagram over Auth

4.4.3 Proxy

For å distribuere systemet er det satt opp en reverse-proxy 3.4.4 i docker nettverket. Dermed går alle forespørsler fra klient-maskiner gjennom proxyen, som bearbeider og videresender forespørselen til den korrekte tjeneren. Klient-maskiner får også svar fra proxyen, som skaper illusjonen at klienten kommuniserer direkte med de interne tjenerene.

Kapittel 5

Diskusjon

Det som skal gjennomgås i dette kapittelet er en drøfting av flere avgjørelser som ble gjort i løpet av prosjeket. Her vil kapittelet dele seg opp i to hoved-tema; teknisk resultat og gjennomførelse av prosjeket.

5.1 Teknisk resultat

I dette underkapittelet blir det tekniske resultatet drøftet, her blir det gjennomgått tanker rundt produktet som ble utviklet, og ting som fungerte godt eller kunne vært gjort annerledes.

5.1.1 Release Note Editor

Den implementerte tekst-editoren for ReleaseNotes 4.2 inkluderer all ønsket funksjonalitet for redigering av Release Notes, men sitter igjen med noen uønskede mangler.

Opplasting av bilder fra URL lagrer ikke bildet på API Serveren på samme måte som opplasting fra lokal maskin. Dette er noe som skulle vært implementer, da man ikke vil inkludere URL fra eksterne nettsider, der man ikke har kontroll på om de blir tilgjengelige i fremtiden.

I tillegg mangler editoren noen form for validering. På grunn implementasjonen av visning av release notes 4.2.7. Den beskrevne klassfiseringen av Release Notes er ikke på noen måte beskrevet for brukeren. Dette er dårlig design, siden man må vite hvordan ting fungerer i backend for å effektivt kunne bruke editoren.

5.1.2 Release Editor

Den implementerte Release Editoren 4.3 fungerer som forventet for oppbygging av Releases, med et intuitivt drag-and-drop system, fullstendig form validering, og nyttige funksjoner som forhåndsvisning og editering av Release Notes.

Et problem som vil oppstå i bruk av Release Editoren, er at den ikke tar hensyn til at det kan finnes mange Release Notes. I nåverende implementasjon vil den naivt vise alle tilgjengelige. Her vil det være nødvendig med for eksempel et paginering-system. En funksjon som derimot på en måte negerer dette, er at det går an å filtre ut Release Notes etter dato.

5.1.3 Adminpanel

Adminpanelets 4.7 nedtrekksmenyer lider av samme mangel som i Release Editor, i at de ikke tar høyde for at der kan finnes store mengder data. Her vil det også være nødvendig med for eksempel et paginering-system. Funksjoner for filtrering ville også vært nyttig her.

Sett bort fra dette, så fungerer funksjonene for administrering av de forskjellige entitetene godt.

Azure-siden 4.8 Adminpanel mangler form-validering. Dette gjør at siden kan oppleves forvirrende, da den noen ganger vil være uten data, uten at brukeren vet hvorfor.

5.1.4 Offentlige sider

Mobilvennlighet er en annen mangel i systemet. Det var et av ønskene i oppgavebeskrivelsen at de offentlige sidene skulle være mobilvennlig. Selv om siden i mange tilfeller fungerer godt på mobil, er det ikke tatt nok hensyn til i utviklingen til å kalle siden mobilvennlig.

5.1.5 Frontend

I denne seksjonen skal resultater rundt implementering av frontend-delen av prosjektet diskuteres.

React

Bruk og læring av React var noe gruppen erfarte positivt. En oversiktlig og dekkende offisiell dokumentasjon med API referanse, og dekning av konsepter og best practices gjorde det raskt å starte utviklingen, med et kunnskapsbasert utgangspunkt. I tillegg har React et stort og aktivt økosystem. Dette gjorde det enklere å løse bugs, og finne ressurser for læring. Men siden det skulle læres et helt nytt rammeverk, oppstod det likevel noen utfordringer.

Det å komme fra en objekt-orientert bagrunn med Java, til å skulle utvikle i React med programmeringsspråket JavaScript var utfordrene i starten. Det krevde tid å mestre bruken av nye konsepter som dynamiske datatyper, arrow functions, immutable data, og callbacks. React er slik at alle komponent-tilstander er immutable, så blir man nødt til å; kopiere tilstanden, gjøre nødvendig manipulering for så å lagre den. Dette er enkelt nok å gjøre for et enkelt objekt eller primitiv, men det ble fort komplisert når det ble snakk om større eller komplekse arrays, der det stilte store krav til forståelse av funksjoner i JavaScript.

En annen utfordring som oppstod var at det finnes to måter å lage React-komponenter; Klasse-baserte og funksjonelle komponenter. Det ble i starten brukt klasse-baserte komponenter, da siden konseptet om klasser var noe gruppen var mer komfortabel med. Det ble derimot etterhvert oppdaget at funksjonelle komponenter ble introdusert av React for å erstatte klasse-baserte komponenter, og at de på flere måter var enklere å arbeide med, og produserte mer lesbar kode. Det ble derfor bestemt å omstille til, og kun bruke funksjonelle komponenter. For å gjøre dette måtte gruppemedlemmene sette seg inn i hvordan man effektivt bruker funksjonelle komponenter. Ett av de store konseptene som måtte læres var react-hooks, som opptok en del av tiden som kunne ha blitt brukt til

å lage nye funksjoner. Samtidig som at det tok litt tid å gjennomføre denne omstillingen, så gjorde gruppen lurt i å ikke implementere disse endringene for hele prosjektet umiddelbart, men tok det heller for alle nye komponenter eller når en eldre komponenet skulle endres på. Selv om dette ble brukt ekstra tid på, gjorde det mange problemer mer overkommelige, og har gitt gruppen en dypere forståelse for både javascript og react.

TypeScript kunne ha blitt brukt istedenfor JavaScript, der TypeScript kunne ha gitt systemet gunstige fordeler når det kommer til å øke stabilitet og avdekke bugs. Et eksempel på dette er static typechecking som setter strengere krav til håndtering med hensyn på datatypene deres. Årsaken til at dette ikke ble brukt, var at TypeScript ble oppdaget som et alternativ på et for sent tidspunkt til å være hensiktsmessig å implementere.

Arkitektur

De store arkitektur-valgene som ble tatt tidlig i prosjektet, viste seg å være lønnsomme, og har bidratt til mer håndterbar kode.

Redux brakte i begynnelsen med seg en stor læringskurve, og introduserte mye boilerplate-kode for tilsynelatende lite funksjonalitet. Over tid og med læring ble det likevel enklere, etter oppdagelse av nye metoder og best practices. Til slutt endte det som et kraftig og fleksibelt system som gjør stor nytte i systemet, ved å bidra til å oppfylle *single source of truth* og *seperation of concerns*. Dette gjør det blant annet lettere å utvide systemet med ny funksjonalitet, eller å endre på implementasjonen til eksisterende funksjonalitet.

Testing

Testing av frontend har mange positive momenter. Det ble implementert automatisk testing med Azure Pipelines, og det ble lært nye test-praksiser. Det kunne likevel blitt lagt mer vekt på testing, da spesielt react komponentene er dårlig dekt av tester.

Automatisk testing med Azure DevOps var et veldig nyttig verktøy. Dette gav stor gevinst for liten innstats, da det kun trengtes å settes opp en gang, og fungerte deretter som forventet resten av prosjektet.

I implementeringen av Redux, ble det brukt en test-først-tilnærming. Dette var nyttig siden ønsket resultat var kjent, men implementasjonen var utsatt for bugs. Dette forenklet implementasjonen og læringsprosessen. Ved å teste Redux-delene i isolasjon, ble de også enklere å implementere.

Test-dekningen av React-komponenter er et område i prosjektet som er ufullstendig. Bortsett fra conformance testing av screens, og noen få unit tester på komponenter, er få komponenter testet. Økt fokus på testing, spesielt i de store komponentene som ReleaseEditor ville vært nyttig, da det ofte var i disse det oppstod problemer.

Generell Design og Brukeropplevelse

Bruken av Material-UI har ført til at brukergrensesnittet er konsistent i design og levende", ved hjelp av forskjellige animasjoner. Brukergrensesnittet er også raskt og reaktivt, noe som delvis kan tilskrives effektiviteten til React, men

også til bruk av Redux og optimaliseringsteknikker. Implementasjonen av Redux sørger for at data som brukes av brukergrensesnittet alltid er oppdatert i henhold til nettverksoperasjoner som blir gjort. Det ble også gjort optimaliseringer i *ReleaseEditor* som gav økt ytelse. Det finnes også generell tilbakemelding i brukergrensesnittet i form av en loading-bar og notifikasjoner, noe som løfter opplevelsen.

5.1.6 Backend

Det som skal drøftes i dette underkapittelet er avgjørelsene som ble gjort i back-end der Arkitekturen, testingen og designvalget blir gjennomgått.

REST API

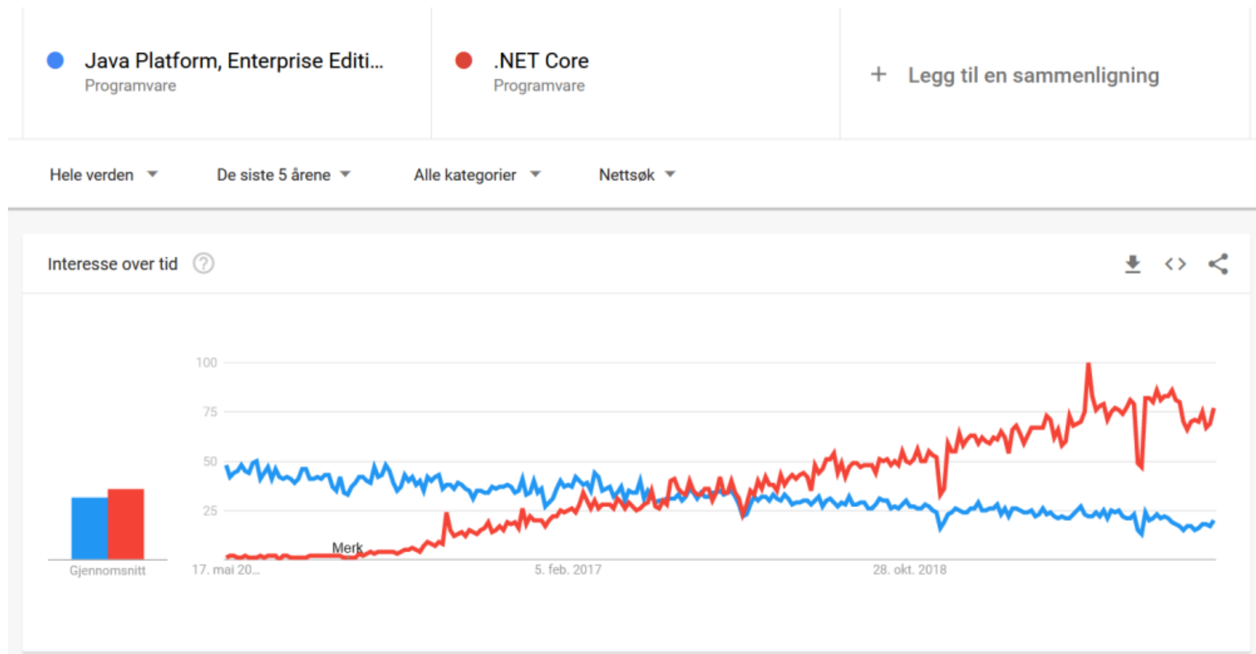
Når det kommer til hvordan gruppen strukturerte og erfaringene med bruk av REST API, så er det stort sett bare positive ting å si. Der det ble fra starten av tatt i bruk flere gode konvensjoner som sørget for en god struktur.

I løpet av prosjektet ble det tidlig tatt i bruk Postman med samarbeidsfunksjonalitet, dette betyr at det ble bygd opp en samling av for eksempel-requester, miljøvariabler og query- og path-parameter. Dette gjorde at når en person hadde lagt inn dette, så var det umiddelbart tilgjengelig for neste mann. En annen fordel med bruken av dette verktøyet var at det var enkelt å få en komplett api-dokumentasjon, siden denne ble generert med utgangspunkt i data som eksisterte fra før.

Endepunktene som ble satt opp tok i bruk konvensjonene nevnt i kap. 3.4.2 som gjorde at det tidlig i prosessen ble en god struktur. Selv om det stort sett ble en god struktur så ble det noen ganger litt vanskelig å bestemme hvordan bruken av query- og path-parameter. Et eksempel på dette er i endepunktet for Mapping, der dagens løsning består av muligheten til å hente ut all mapped fields med queryparam `mapped`. Dette er noe som like gjerne kunne ha blitt gjort ved å inkludere `mapped` som et path-parameter.

ASP.NET Core

Erfaringene gruppen hadde med dette rammeverket var stort sett positivt det finnes veldig mye god dokumentasjon og eksempler på hvordan man setter opp en slik applikasjon. Det rammeverket gruppen tidligere har erfaring med er rammerverket Java EE, som i motsetning til ASP.NET Core var et rammeverk som opplevde var mye vanskeligere å sette seg inn i, der miljøet som bruker ASP.NET Core er større enn Java EE. se figur 5.1



Figur 5.1: Oversikt som viser popularitet for Java EE og .NET Core på google over de fem siste årene.

Arkitektur

Når det kommer til arkitekturen i back-end så har gruppen tatt utgangspunkt i flere design-mønstre som repository-pattern og single source of truth, som har tilsynelatende har gjort at utviklingsprosessen ble oversiktlig og gjennomførbar. En ting som gruppen bet seg merke i var at aldri var noe problem å finne frem i koden underveis, der det til en hver tid var åpenbart hvor mange skulle begynne å lete etter noe som for eksempel et repository eller en modell.

Det var også noe gruppen opplevde som negativt med valgt arkitektur, dette var at det ble skrevet mye boilerplate-kode for å oppretteholde arkitekturen. Dette irritasjonsmomentet ble veldig tydelig når veldig enkle funksjoner skulle inkludereres i systemet, der for eksempel hvis en skal opprette et endepunkt som bare skal hente ut en liste av en ny entitet, så blir man nødt til å lage heile løypen med Controller, Service, Repository, Model, Response og Resource som forklares i kap 3.4.2. Selv om det kan føles bortkastet å skrive all denne boilerplate-koden, så gjør det jobben enklere med tanke på oversikt og videreutvikling. Hvis man fortsetter eksempelet og skal opprette flere operasjoner for samme entitet, så trenger utvikleren bare å opprette et par nye metoder i de klassene som allerede opprettet. Samtidig som at en annen utvikler for prosjektet vet akkurat hvor man skal begynne å lete etter spesifikke funksjoner.

Database

Erfaringen til gruppen med tanke databasens design og rammerverk har vært stort sett positive. Gruppen har tidligere erfaring med bruk av SQL og rammerverk som Hibernate se kap 2.13.8, der EF-Core kommer ut som det foretrukne alternativet.

Måten man bygger opp en databasen på gjennom naming-conventions se kap 2.13.7 og hente ut data uten å måtte skrive mye boilerplate-kode, har opplevdes som enkelt og oversiktlig. Når det kommer til måten man kan hente ut data på uten å ta hensyn til sirkulære-avhengigheter slik som tidligere erfaringer med Hibernate. Dette har gjort at det har blitt brukt minimalt med slike trivielle problem.

Hvis man setter opp EF-Core mot å skrive alt av SQL for hånd, så går det raskere og gir en innebygd fordel når det kommer til oversikt. Gruppen har tidligere erfart at det stiller store krav til utvikleren hvis man skal skrive alt av SQL-kode selv, der det fort blir uoversiktlig hvis man ikke jobber strukturert, der får man en gratis fordel med EF-Core der utvikler blir anbefalt en måte å strukturere koden på som også inneholder minimalt med boilerplate-kode.

Designet med tanke på datamodellen har vært grei å jobbe med, der det bare finnes et par tilfeller der endringer måtte inn. Det oppstod to situasjoner hvor det måtte endres i strukturen. Dette var i begge tilfellene på grunn av manglende many-to-many-forhold. En svakhet i den endelige databasen er at *ReleaseNote* tabellen inkluderer overflødig informasjon om work item. Dette er informasjon som kun brukes av *ReleaseNoteEditor*, og skulle hatt sin egen tabell.

Sikkerhet

Den endelige sikkerhetsløsningen oppfyller alle krav stilt til oppgaven, der det i tillegg er tatt hensyn til videre utvikling. En måte dette ble tatt hensyn til for videre utvikling var gjennom å implementere et rolle-basert autoriseringssystem, som tillater utvidelse med flere roller hvis behovet skulle oppstå.

En ting som mangler i denne løsningen med tanke på sikkerhet en måte å bevare en private-key på ved stenging av applikasjonsserveren. Løsningen som finnes idag generer en tilfeldig private-key ved oppstart, og eksiterer i minnet helt til den blir slått av. Dette er en upraktisk ordning som gjør at alle må logge på igjen. En annen sak som mangler er automatisk fornyelse og oppsigelse av token i front-end. I denne løsningen finnes logikken i back-end for å gjøre akkurat dette, men det ble en funksjon som var "nice to have".

Testing

Når det kommer til testing av back-end så hadde gruppen integration-testing for å validere at alle endepunktetene oppførte seg som forventet. Dette er gruppen ikke fullt så fornøyd med grunnet at de var tungvint å eksekvere test-runner for hver gjennomgang. Disse testene kunne til fordel blitt automatisert til å kjøres på alle nye pull requests slik som testene i frontend.

Gruppen innså etter hvert i prosjektet at det hadde vært nyttig med unit-testing i back-end. Med unit-testing i tillegg til integration-testing, så hadde det sannsynligvis gått enda fortere å finne eventuelle feil. En annen ting som kunne ha skjedd er at utviklingen per ny funksjon kunne gått fortere om det fantes lette unit-tester som går fort å kjøre, da det kan ta lang tid å gjøre gjennom integration-tester.

Selv om det finnes flere fordeler med å bruke unit-tester, så har det gått greit å utvikle i back-end, der det har vært få

bugs og når det har vært bugs så har det gått relativt kort tid å løse. Årsaken til at det har fungert bra uten unit-tester er vanskelig å bevise. Det kan være flere årsaker for dette; kompleksiteten på business logikken har som regel vært lav, god kodestil med tanke på tidligere nevnt arkitektur og design patterns, eller det kan være ren tilfeldighet.

5.1.7 Eksisterende Løsninger

Det ble gjort en undersøkelse på om det eksisterer noen lignende system, der det ble satt fokus på om disse systemene oppfyller samme krav som oppdragsgiver stilte for denne oppgaven. Resultatet av denne undersøkelsen var at det ikke fantes noen system som gjorde akkurat det samme.

Et system som oppfyller deler av kravet er Release Notes IO se lenke [63]. Der dette systemet tilbyr mange flotte funksjoner for å legge ut release notes med funksjoner som; lest release note, responsivt design og mail-varsel. Det dette systemet mangler er hele den delen som omfatter automatisering i den grad oppgaven stiller krav om.

Et annet system som lignet på det denne oppgaven skal løse var Fastlane se lenke [27]. Dette systemet ser ut til å ha mer fokus på app-utvikling, der det finnes flere funksjoner som integrering mot app-store og automatisk screenshot-mulighet. Dette systemet ble for spesifikt for app-utvikling og var ikke helt egnet til oppgaven.

5.2 Gjennomførelese av prosjektet

Her drøfter gruppen hvordan bruk av samarbeidsteknikker og verktøy har fungert, hva som gikk bra, hva som kunne forbedres, og lærdom fra prosjektet.

5.2.1 Smidig utvikling

Smidig utvikling med Scrum har i de fleste tilfeller gått fint, men med noen mangler i utførelsen av prosessen. Samtaler med oppdragsgiver indikerer at oppdragsgiver også er fornøyd med samarbeidet. Videre drøftes Scrum momenter i mer detalj.

Dette prosjektet utnevnte aldri en produkteier. Årsaken til at det ikke ble utnevnt en produkteier var på grunn av manglende kunnskap om Scrum-roller ved prosjektstart. Rollen og rollens ansvar har falt mer på gruppen selv, men til tross for mangel på produkteier så har gruppen klart å følge Scrum nokså godt.

Noe som helt klart kunne blitt forbedret er valget om når user storys blir estimert. Ved flere anledninger dukket det opp et ønske om en funksjonalitet under møte, som også ble estimert under møte. I retrospekt så ser gruppen på dette som ufornuftig bruk av oppdragsgivers tid, og dårlig estimering av user story. Estimering av user storys fikk aldri et fast sted i Scrum strukturen.

Det som kan skje i en utviklingsprosess er at både utviklingsteam og oppdragsgiver forstår produktet bedre etter at det har begynt å ta form. Dette gjør at det kan bli variasjoner mellom påtenkt produkt og sluttprodukt. I vårt tilfelle

ble det et større system enn først påtenkt, der flere funksjoner ble lagt til underveis. Slike funksjoner er blant annet ReleaseNoteEditor4.2 og Mapping4.1.4.

Sprint-møter

Møter ble brukt for å vise framgang i prosjektet og for å planlegge videre framgang. Disse møtene forsikret at oppdragsgiver fikk innsikt i hva som var utviklet, og være med på å bestemme videre prioriteringer. På dette viset fikk oppdragsgiver både oversikt og mulighet til å forme prosjektet etter sine egne ønsker.

Sprint Retrospectives

Gruppen er stort sett fornøyd med sprint-retrospectives som ble lagd, og anser de som et nyttig verktøy for å påbygge gode vaner og forebygge dårlige. De siste sprint-retrospektivene var litt utfordrende med tanke på å finne nye momenter å ta med grunnet lavere arbeidsmengde og mer fokus på bugs og rapportskriving.

Scrum-board

Scrum-board ble brukt for å holde oversikt over hvilke oppgaver som finnes og tilstanden deres. Det å lage gode tasks viste seg å ha en viss læringskurve. Det mest vanlige eksempelet på dårlige tasks er at gruppen opplevde er dissonans mellom hva taskens påtenkte oppgave var, og hva utvikleren som valgte å gjøre oppgaven gjorde.

Dette kan resultere i at det blir lagd mye mer enn nødvendig. Dermed er det ikke lengre tydelig hva som blir gjort basert på taskens tekst.

En annen mulighet er at noe annet enn hva tasken var påtenkt ble lagd. Noe som resulterer i bortkastet arbeid.

Disse erfaringene gjorde at gruppen lagde normer og regler for oppbygging av tasks. Dette resulterte i mer tydelige og spesifikke tasks.

5.2.2 Versjonskontroll

Pull Request og Code Review

All kode gjennomgikk henholdvis gjennom pull request og code review. Dette gav flere positive virkninger i prosjektet.

Code reviews avdekte bugs, forbedringer til implementasjon og alternative implementasjoner. Dette førte til økt forståelse blant gruppemedlemmene og penere kildekode.

I begynnelsen av prosjektet inkluderte pull requests som oftest ikke en beskjed som utdypte hva som ble endret eller lagt til. Dette ble mer vanlig lengre inn i prosjektet. Dette opplever gruppen som positivt siden man får presentert problemstillingen på en tydelig måte, som hjelper å leseren å sette seg inn i hva og hvorfor kode endres eller blir lagt til.

En annen viktig fordel med denne teknikken var at alle i utviklingsteamet fikk innblikk i hva som befant seg i prosjektets kildekode, slik at alle satt igjen med en grunnleggende oversikt over hvordan systemets byggeklosser ser ut.

Alt i alt bidro pull requests og code reviews til en mer strukturert arbeidsmåte, og økt kvalitet i det endelige produktet.

Kapittel 6

Konklusjon

Det som skal gjennomgås i dette kapittelet er en oppsummering av hva som gjenstår å gjøre med systemet, og selve konklusjonen av oppgaven.

6.1 Resultatet

Cordel ønsket seg et system som forenklet prosessen å dokumentere nye funksjoner i releases av produktene deres. Vi har utviklet en full-stack løsning som svarer på mange av ønskene. Vi har fått muligheten til å utdype oss i nye teknologier og rammeverk. Fokuset gjennom oppgaven har vært på arkitektur med tanke på utvidelse og videreutvikling, og samarbeid med oppdragsgiver for å utvikle et tilfredsstillende produkt.

Gjennom prosjektet har gruppen fått en grundig forståelse for hvordan full-stack-applikasjoner bygges opp. Der gruppen fikk oppleve mange design-mønstre, biblioteker og rammeverk med sine styrker og svakheter. Styrkene til disse design-mønstrene har bidratt til et prosjekt med kode som er strukturert og utvidbar. Svakhetene har gitt god lærdom, og tas med videre til betraktning i utvikling av fremtidige prosjekter.

Gruppen klarte å opprette en web-applikasjon som gjorde det effektivt å dokumenterte og publisere releases gjennom tilpassede redigeringsverktøy. Denne webapplikasjonen inneholdte også fullstendige verktøy for å gjøre administrative handlinger på data. Alt dette er pakket i et raskt, reaktivt og moderne brukergrensesnitt.

Målet om å integrere systemet med Azure var ganske vagt definert i oppstarten av prosjektet, der oppdragsgiver ønsket seg funksjoner som kunne hente data fra Azure og bruke i dette systemet. Det gruppen klarte å lage inkluderer importering av releases fra Azure Pipelines, samt mapping-systemet. Selv om integreringen er begrenset til disse funksjonene, er det en god begynnelse som gir ekstra verdi ved bruk av produktet.

Som tidligere diskutert så er gruppen fornøyd med sluttresultatet til back-end. Gruppen endte opp med en back-end med fullt dokumenterte endepunkter etter REST, og en implementasjon som er fleksibel og utvidbar.

Når det kom til planleggingen av prosjektet så stemte overraskende nok den initielle planlegging godt med det som faktisk hadde skjedd, der gruppen klarte å fullføre MVP innen planlagt tidsramme og utførelse av brukertesting.

Alt i alt har det vært en krevende prosess, med stort læringsutbytte, både innen teknologi, og ved utvikling av et produkt med smidig metodikk, der produkt og krav hele tiden utvikler seg. Det endelige produktet inkluderer funksjoner som oppfyller initielle krav og målsetninger, og har også stort rom for å vokse.

6.1.1 Gjenstående arbeid

Når det kommer til videre arbeid finnes det flere små-mangler som kan fikses på. Disse inkludere blant annet, validering på et skjema og konsistens bruk av farger.

Manglene som ble dekt i kapitlene 5.1.1, 5.1.2, 5.1.3, og 5.1.4 er et godt utgangspunkt for videre utvikling, da mange av disse er mangler som vil hindre produktiviteten av produktet.

Noe som ville økt nytten av systemet er å utvide integrasjonen med Azure. Dette kunne for eksempel være automatiske funksjoner der Release Note Systemet automatisk oppdager endringer i Azure.

Det vil trenges noe finpuss på de offentlige sidene for å gjøre dem mer presentable, mobilvennlige og konsistent i stil med Cordel sin webside.

Når produktet blir tatt i reell bruk, vil det også komme frem hva som er mest naturlig å arbeide med videre.

6.2 Samarbeid

Det har allerede blitt diskutert hvordan smidig utviklingsmetodikk har blitt brukt gjennom prosjektet, og hvilke metoder som ble brukt. Det gruppen tenker om gjennomførelsen er at man aldri blir ferdig å forbedre seg som en gruppe. Kontinuerlig forbedring både i software og teamet er løsningen på et godt utviklingsprosjekt.

Appendices

A WebAPP Kildekode -> ReleaseNotes-WebApp.zip

B WebAPI Kildekode -> ReleaseNotes-WebApi.zip

C Proxy Kildekode -> ReleaseNotes-Proxy.zip

D Forprosjekts-rapport -> forprosjektrapport_bacheloroppgave.pdf

E Wireframes -> Wireframes.pdf

F Code coverage Rapport -> code_coverage.pdf

G API documentation -> apidoc.pdf

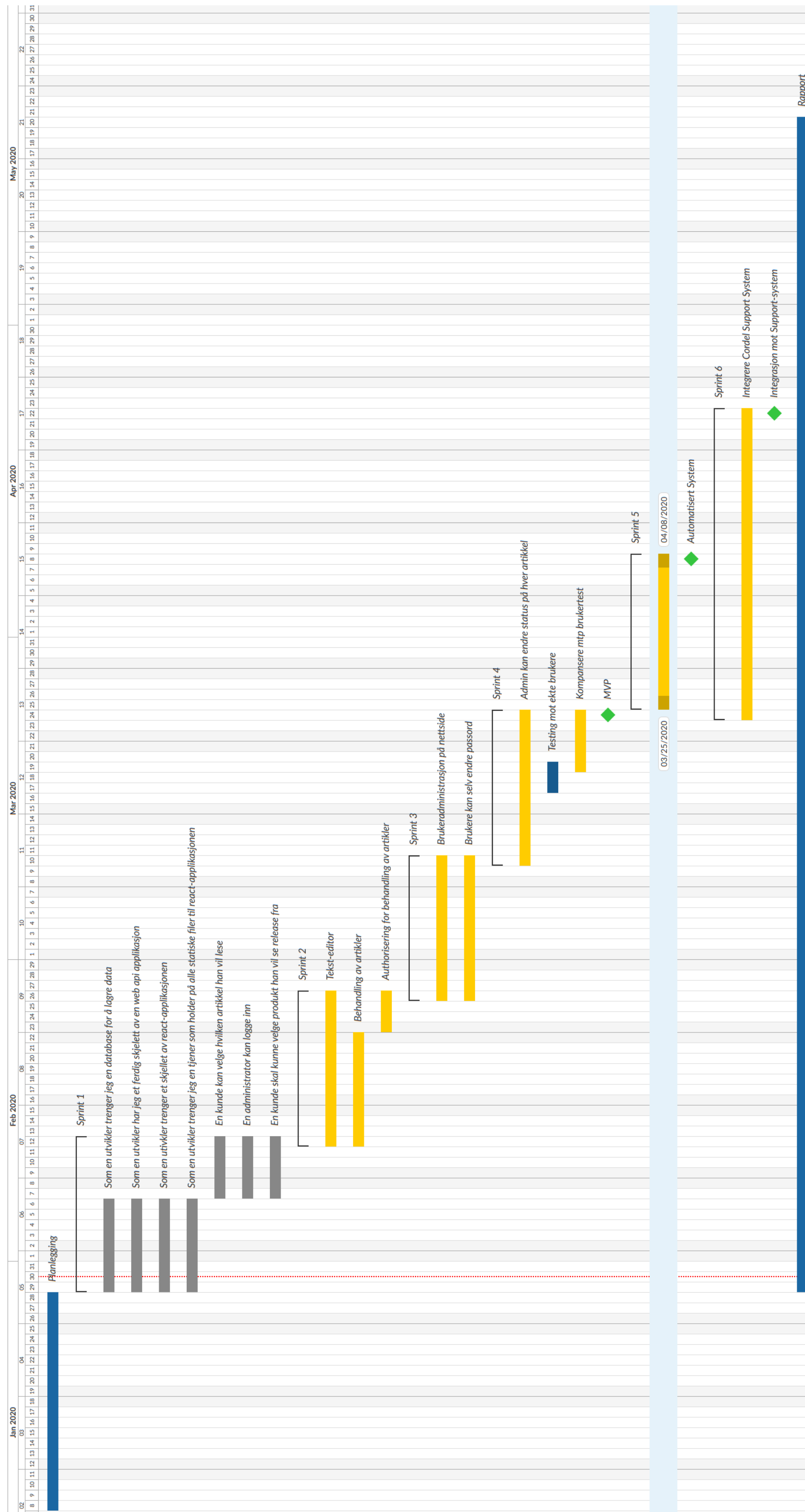
H Brukermanual -> forprosjektrapport_bacheloroppgave.pdf

I Sprint retrospectives -> retrospectives.pdf

J Logger -> loggs.pdf

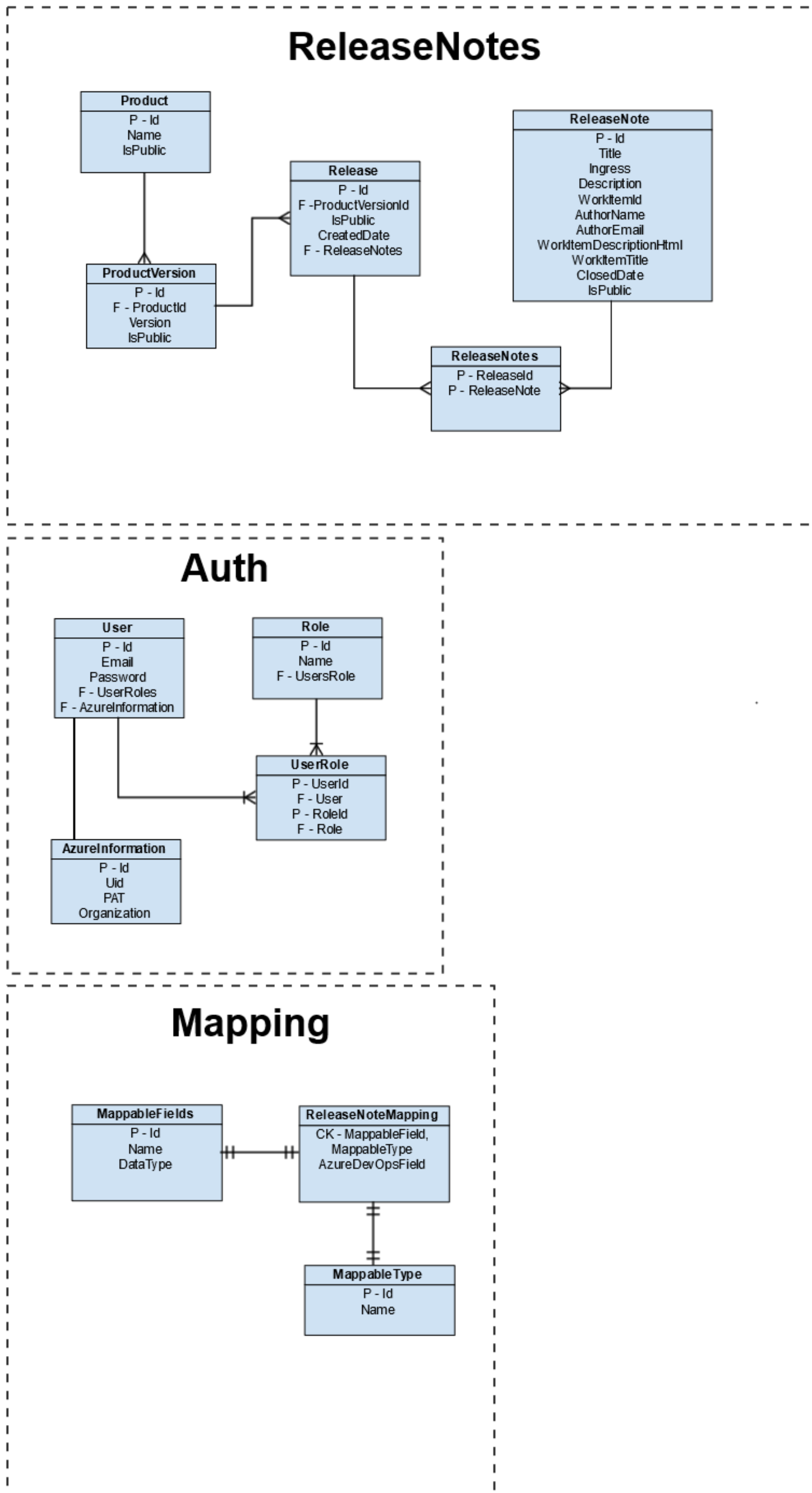
K Sprint reviews -> sprintReviews.pdf

L



Fullstendig Gantt-Diagram.

M



Fullstendig ER-Diagram.

N

ReleaseNoteController
Class
Controller
- Fields
  - _mapper
  - _releaseNoteService
- Methods
  - CreateReleaseNoteAsync
  - CreateReleaseNoteFrom...
  - GetAllAsync
  - GetSpecificReleaseNote
  - ReleaseNoteController
  - RemoveReleaseNote
  - UpdateReleaseNoteAsync

ImagesController
Class
Controller
- Fields
  - _imageService
  - _mapper
- Methods
  - GetImage
  - ImagesController
  - UploadImage

ProductVersionsController
Class
Controller
- Fields
  - _mapper
  - _productVersionService
- Methods
  - GetAllAsync
  - ProductVersionsController

UsersController
Class
Controller
- Fields
  - _mapper
  - _userService
- Methods
  - ChangeUserPassword
  - CreateUserAsync
  - GetAzureInformation
  - ListAsync
  - UpdateUser
  - UsersController

MappableFieldsController
Class
Controller
- Fields
  - _mappableService
  - _mapper
- Methods
  - ListAsync
  - MappableFieldsController
  - UpdateReleaseNoteMap...

ReleasesController
Class
Controller
- Fields
  - _mapper
  - _releaseService
- Methods
  - CreateFromWorkItems
  - GetAsync
  - ListAsync
  - PostAsync
  - PutAsync
  - ReleasesController
  - RemoveRelease

LoginController
Class
Controller
- Fields
  - _authenticationServ...
  - _mapper
- Methods
  - GetPublicKey
  - LoginAsync
  - LoginController
  - RefreshTokenAsync
  - RevokeToken

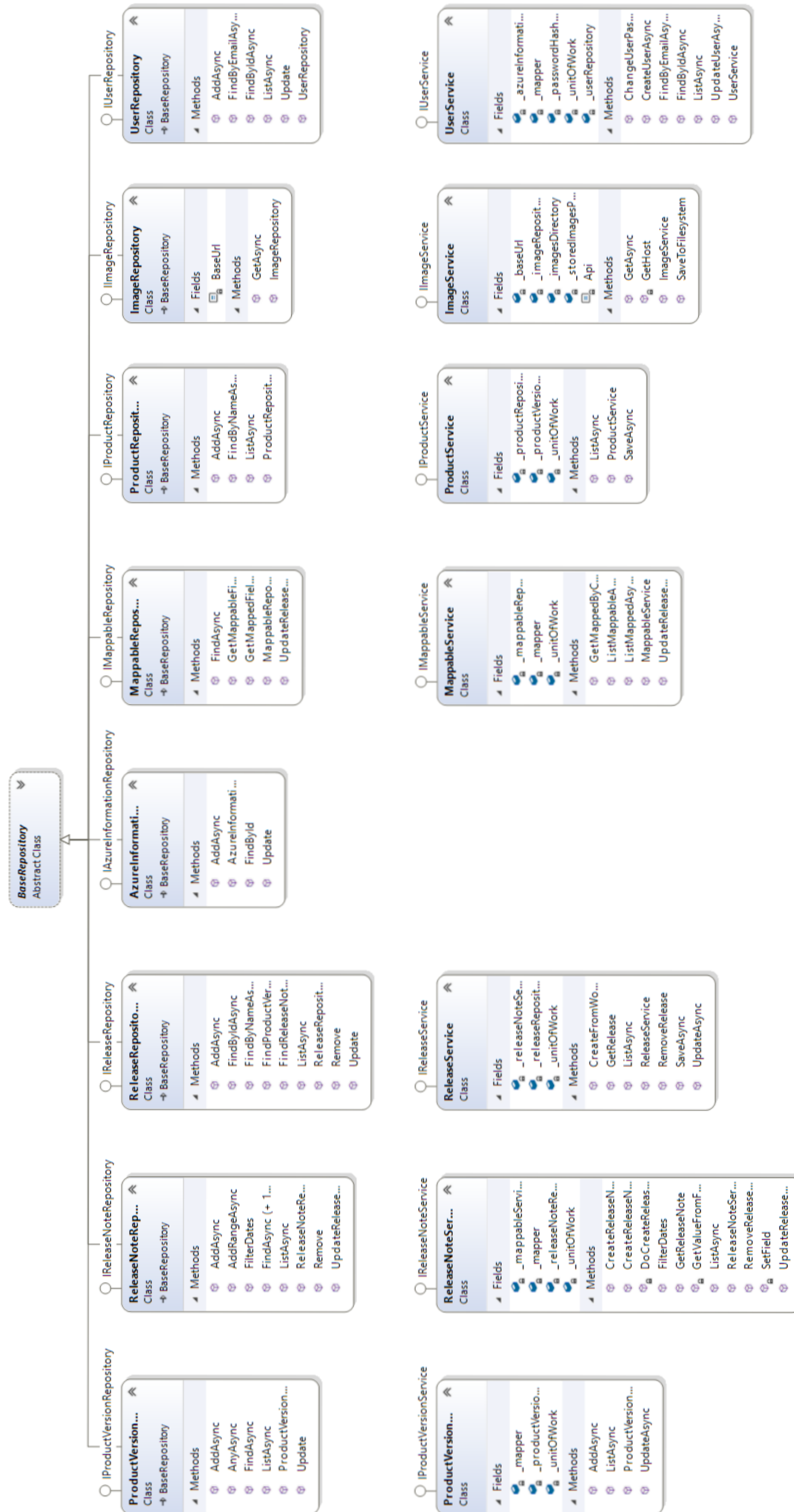
ProductsController
Class
Controller
- Fields
  - _mapper
  - _productService
  - _productVersionService
- Methods
  - AddVersionAsync
  - CreateProductAsync
  - GetAllAsync
  - ProductsController
  - UpdateVersionAsync

Fullstendig ER-Diagram.

O

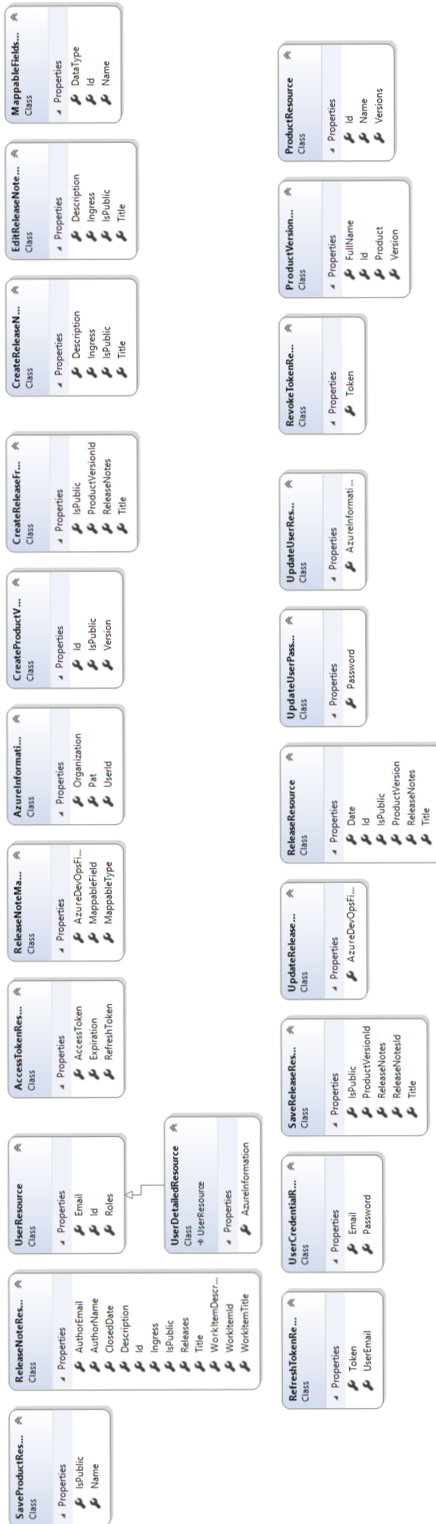
Klassediagram med fullstendig oversikt over Modell-klassene.

P



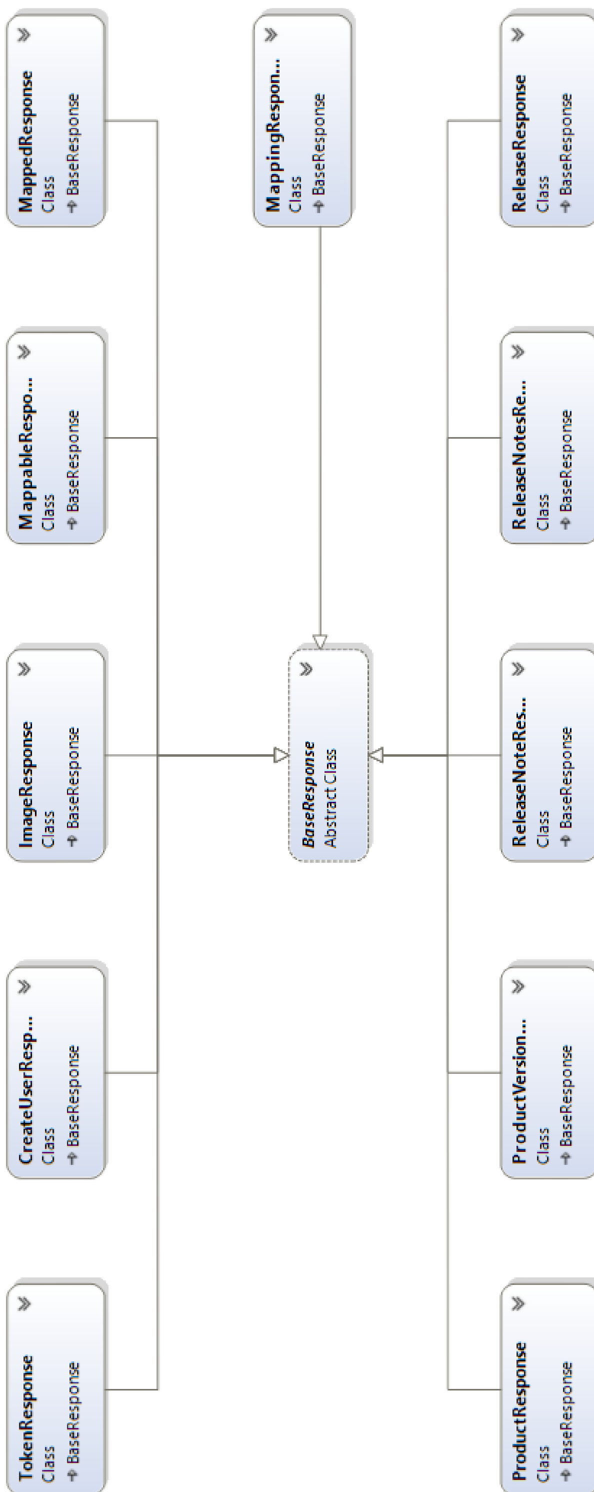
Klassediagram med fullstendig oversikt over klassesene og interfacene til Repository og Service.

Q



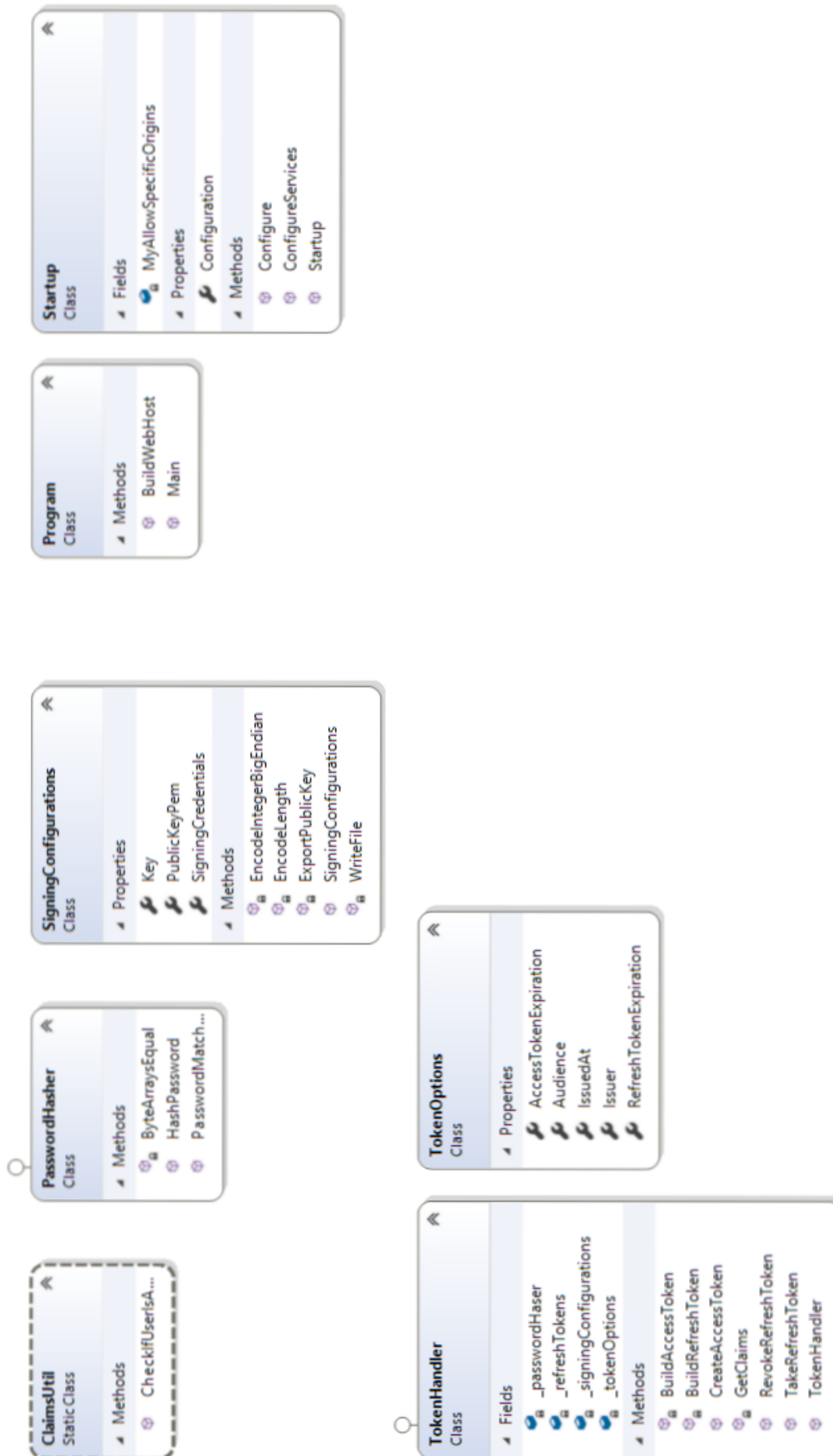
Klassediagram med fullstendig oversikt over Resource.

R



Klassediagram med fullstendig oversikt over Response.

S



Klassediagram med fullstendig oversikt over klassene som omhandler sikkerhet, startup og program.

Bibliografi

[1] About pull requests - GitHub Help, May 2020. URL https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests. [Online; accessed 19. May 2020].

[2] Callback function, May 2020. URL https://developer.mozilla.org/en-US/docs/Glossary/Callback_function. [Online; accessed 19. May 2020].

[3] Certification & conformity, May 2020. URL https://www.iso.org/conformity-assessment.html. [Online; accessed 17. May 2020].

[4] Tailwind CSS - A utility-first CSS framework for rapidly building custom designs, May 2020. URL https://tailwindcss.com. [Online; accessed 19. May 2020].

[5] Rick Anderson and Steve Smith. Asp.net core middleware, APR 2020. URL https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1. [misc; accessed 4. May. 2020].

[6] Arash. Interface Driven Development (part 1) - Arash - Medium. *Medium*, May 2018. URL https://medium.com/@aaraashkhan/interface-driven-development-part-1-b7c7011860c0.

[7] ardalis. Dependency injection into controllers in ASP.NET Core, May 2020. URL https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-3.1#constructor-injection. [Online; accessed 16. May 2020].

[8] atlassian. react-beautiful-dnd, Apr 2020. URL https://github.com/atlassian/react-beautiful-dnd. [Online; accessed 14. May 2020].

[9] Atlassian. Agile Scrum Roles | Atlassian, May 2020. URL https://www.atlassian.com/agile/scrum/roles. [Online; accessed 16. May 2020].

[10] AutoMapper. AutoMapper, Mar 2019. URL https://automapper.org. [Online; accessed 18. May 2020].

[11] axios. axios, May 2020. URL https://github.com/axios/axios#interceptors. [Online; accessed 15. May 2020].

[12] AzureDevopsServices. Azure DevOps Services | Microsoft Azure, May 2020. URL https://azure.microsoft.com/en-us/services/devops/#Customer. [Online; accessed 17. May 2020].

[13] Chai. Chai, Jan 2020. URL https://www.chaijs.com. [Online; accessed 14. May 2020].

[14] Mike Cohn. What Are Story Points? *Mountain Goat Software*, Aug 2019. URL https://www.mountaingoatsoftware.com/blog/what-are-story-points.

[15] Mike Cohn. Planning Poker: An Agile Estimating and Planning Technique, May 2020. URL https://www.mountaingoatsoftware.com/agile/planning-poker. [Online; accessed 15. May 2020].

[16] Mike Cohn. User Stories and User Story Examples by Mike Cohn, May 2020. URL https://www.mountaingoatsoftware.com/agile/user-stories. [Online; accessed 15. May 2020].

[17] Contributors to Wikimedia projects. Separation of concerns - Wikipedia, May 2020. URL https://en.wikipedia.org/w/index.php?title=Separation_of_concerns&oldid=956142286. [Online; accessed 13. May 2020].

[18] Contributors to Wikimedia projects. Memoization - Wikipedia, Mar 2020. URL https://en.wikipedia.org/w/index.php?title=Memoization&oldid=948331541. [Online; accessed 19. May 2020].

[19] Flavio Copes. Styled Components, May 2020. URL https://flaviocopes.com/styled-components. [Online; accessed 14. May 2020].

[20] CreateReactApp. Create React App · Set up a modern web app by running one command., Mar 2020. URL https://create-react-app.dev/docs/getting-started. [Online; accessed 14. May 2020].

[21] DeepScan, 2020. URL https://deepscan.io/home/. [misc; accessed 04. May. 2020].

[22] Docker. Docker Engine overview, May 2020. URL https://docs.docker.com/engine. [Online; accessed 14. May 2020].

[23] DockerCompose. Overview of Docker Compose, May 2020. URL https://docs.docker.com/compose. [Online; accessed 14. May 2020].

[24] DockerContainer. What is a Container? | Docker, May 2020. URL https://www.docker.com/resources/what-container. [Online; accessed 13. May 2020].

[25] EfCoreNamingConventions. Entity Framework Core Conventions, May 2020. URL https://www.entityframeworktutorial.net/efcore/conventions-in-ef-core.aspx. [Online; accessed 18. May 2020].

[26] facebook. draft-js, May 2020. URL https://github.com/facebook/draft-js. [Online; accessed 14. May 2020].

[27] Fastlane. fastlane - App automation done right, May 2020. URL https://fastlane.tools. [Online; accessed 19. May 2020].

[28] FunctionAndClassComponents. Function and Class Components, May 2020. URL https://reactjs.org/docs/components-and-props.html#function-and-class-components. [Online; accessed 16. May 2020].

[29] Evandro Gomes. An awesome guide on how to build restful apis with asp.net core, 2019. URL https://www.freecodecamp.org/news/an-awesome-guide-on-how-to-build-restful-apis-with-asp-net-core-87b818123e28/

[30] GuerillaTesting. What is guerrilla testing and how do you use it?, Sep 2017. URL https://www.userzoom.com/blog/what-is-guerrilla-testing-and-how-do-i-use-it. [Online; accessed 14. May 2020].

[31] Hibernate. Your relational data. Objectively. - Hibernate ORM, May 2020. URL https://hibernate.org/orm. [Online; accessed 18. May 2020].

[32] HistoryAPI. History api, May 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/History_API. [Online; accessed 8. May 2020].

[33] Jest. Jest · xn–fz7h Delightful JavaScript Testing, May 2020. URL https://jestjs.io. [Online; accessed 14. May 2020].

[34] KathrynEE. Using work items to track user stories, & more - Azure Boards and TFS, May 2020. URL https://docs.microsoft.com/en-us/azure/devops/boards/work-items/about-work-items?view=azure-devops&tabs=agile-process#work-item-types-wits. [Online; accessed 13. May 2020].

[35] Eric Kollegger. What is Axios.js and why should I care? - Eric Kollegger - Medium. *Medium*, May 2018. URL https://medium.com/@MinimalGhost/what-is-axios-js-and-why-should-i-care-7eb72b111dc0.

[36] LabUsabilityTesting. Unboxing: laboratory usability testing | Insight | Box UK, May 2020. URL https://www.boxuk.com/insight/unboxing-laboratory-usability-testing. [Online; accessed 14. May 2020].

[37] ManyToMany. Many-to-many relationships, May 2020. URL https://fmhelp.filemaker.com/help/18/fmp/en/index.html#page/FMP_Help%2Fmany-to-many-relationships.html%23. [Online; accessed 17. May 2020].

[38] Jacob Thornton Mark Otto and Bootstrap contributors. Bootstrap, May 2020. URL https://getbootstrap.com. [Online; accessed 19. May 2020].

[39] MaterialDesign. Overview, May 2020. URL https://material.io/design/material-theming/overview.html#material-theming. [Online; accessed 14. May 2020].

[40] MateriUI. Material-UI: A popular React UI framework, May 2020. URL https://material-ui.com. [Online; accessed 14. May 2020].

[41] Mergify, 2020. URL https://mergify.io/. [misc; accessed 04. May. 2020].

[42] Microservices. What are Microservices? | API Basics | SmartBear, May 2020. URL https://smartbear.com/solutions/microservices. [Online; accessed 13. May 2020].

[43] Mocha. Mocha - the fun, simple, flexible JavaScript test framework, May 2020. URL https://mochajs.org. [Online; accessed 14. May 2020].

[44] Justin Morales. Remote usability testing, 2020. URL https://xd.adobe.com/ideas/process/user-testing/remote-usability-testing/.

[45] Mozilla. HTTP response status codes, May 2020. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. [Online; accessed 17. May 2020].

[46] N/A. Introduction to json web tokens, N/A. URL https://jwt.io/introduction/.

[47] N/A. Stateless authentication (token-based authentication), N/A. URL https://doubleoctopus.com/security-wiki/network-architecture/stateless-authentication/.

[48] Neoteric. Single-page application vs. multiple-page application. *Medium*, Jun 2018. URL https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.

[49] OneToMany. One-to-many relationships, May 2020. URL https://fmhelp.filemaker.com/help/18/fmp/en/index.html#page/FMP_Help%2Fone-to-many-relationships.html%23. [Online; accessed 17. May 2020].

[50] OneToOne. One-to-one relationships, May 2020. URL https://fmhelp.filemaker.com/help/18/fmp/en/index.html#page/FMP_Help%2Fone-to-one-relationships.html%23. [Online; accessed 17. May 2020].

[51] George Ornbo. Linux and Unix exit code tutorial with examples. *George Ornbo*, Nov 2019. URL https://shapeshed.com/unix-exit-codes.

[52] PathParam. Step 3: Parameters (API reference tutorial), May 2020. URL https://idratherbewriting.com/learnapidoc/docapis_doc_parameters.html#path_parameters. [Online; accessed 17. May 2020].

[53] Postman. The Postman API Platform, May 2020. URL https://www.postman.com/api-platform/?utm_source=www&utm_medium=home_hero&utm_campaign=button. [Online; accessed 16. May 2020].

[54] Ed Putans. Single Source of truth and applying it software development. *Medium*, Oct 2018. URL https://medium.com/@edunceputans/single-source-of-truth-and-problems-with-implication-in-an-organisation-588883492133.

[55] QueryParam. Step 3: Parameters (API reference tutorial), May 2020. URL https://idratherbewriting.com/learnapidoc/docapis_doc_parameters.html#query_string_parameters. [Online; accessed 17. May 2020].

[56] React. React – A JavaScript library for building user interfaces, May 2020. URL https://reactjs.org. [Online; accessed 13. May 2020].

[57] React. The data flows down, 2020. URL https://reactjs.org/docs/state-and-lifecycle.html#the-data-flows-down.

[58] react-testing library. Introduction, 2019. URL https://testing-library.com/docs/intro.

[59] ReactRedux. React Redux · Official React bindings for Redux, May 2020. URL https://react-redux.js.org. [Online; accessed 14. May 2020].

[60] ReactRouter. React Router: Declarative Routing for React, Apr 2020. URL https://reacttraining.com/react-router. [Online; accessed 14. May 2020].

[61] ReduxOfficialSite. Redux - A predictable state container for JavaScript apps. | Redux, Apr 2020. URL https://redux.js.org. [misc; accessed 27. Apr. 2020].

[62] ReduxToolkit. Redux Toolkit | Redux Toolkit, May 2020. URL https://redux-toolkit.js.org. [Online; accessed 14. May 2020].

[63] ReleaseNotesIo. Product Release Notes Software & Changelog Tool, May 2020. URL https://releasenotes.io/#features. [Online; accessed 19. May 2020].

[64] Rest. REST API Tutorial, May 2020. URL https://restfulapi.net. [Online; accessed 13. May 2020].

[65] ReverseProxy. What is a Reverse Proxy Server? | NGINX, May 2020. URL https://www.nginx.com/resources/glossary/reverse-proxy-server. [Online; accessed 15. May 2020].

[66] Mark Richards. Software architecture patterns, 2016. URL https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html.

[67] Rick-Anderson. Dependency injection in ASP.NET Core, May 2020. URL https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1. [Online; accessed 13. May 2020].

[68] rowanmiller. Overview of Entity Framework Core - EF Core, May 2020. URL https://docs.microsoft.com/en-us/ef/core. [Online; accessed 14. May 2020].

[69] Scrum. Hva er egentlig Scrum?, May 2020. URL https://www.glasspaper.no/artikkel/hva-er-egentlig-scrum. [Online; accessed 15. May 2020].

[70] Jon P Smith. Part 2: Handling data authorization in ASP.NET Core and Entity Framework Core – The Reformed Programmer, May 2020. URL https://www.thereformedprogrammer.net/part-2-handling-data-authorization-asp-net-core-and-entity-framework-core. [Online; accessed 8. May 2020].

[71] SprintMeetings. What are Agile Scrum meetings?, May 2020. URL https://yodiz.com/help/agile-scrum-meetings. [Online; accessed 15. May 2020].

[72] Sprint/ProductBacklogs. Sprint Backlog vs Product Backlog: What is the Difference? - Cuspy Software, May 2018. URL https://cuspy.io/blog/what-is-the-difference-between-sprint-backlog-and-product-backlog. [Online; accessed 15. May 2020].

[73] tdykstra. Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10), Apr 2020. URL https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application. [misc; accessed 27. Apr. 2020].

[74] TypeScript. TypeScript - Overview - Tutorialspoint, May 2020. URL https://www.tutorialspoint.com/typescript/typescript_overview.htm. [Online; accessed 13. May 2020].

[75] Velocity. What is Velocity in Scrum?, May 2020. URL https://www.visual-paradigm.com/scrum/what-is-scrum-velocity. [Online; accessed 18. May 2020].

[76] W3Schools. HTTP Methods GET vs POST, May 2020. URL https://www.w3schools.com/tags/ref_httpmethods.asp. [Online; accessed 17. May 2020].

[77] wadepickett. ASP.NET documentation, May 2020. URL https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1. [Online; accessed 14. May 2020].

[78] Patrick Westerhoff. What are services and why add them in ASP.NET Core?, May 2020. URL https://stackoverflow.com/questions/55845137/what-are-services-and-why-add-them-in-asp-net-core/55845279#55845279. [Online; accessed 16. May 2020].

[79] WhatsAHook. What's a Hook?, May 2020. URL https://reactjs.org/docs/hooks-state.html#whats-a-hook. [Online; accessed 16. May 2020].

Randa Pedersen Ous

Release Note System

NTNU
Kunnskap for en bedre verden