Lasse Kregnes Hansen
Pål Nyseth
Martin Håhjem Årdal

Informasjonsskjerm for sjakkturneringer

Bacheloroppgave i Dataingeniør
Veileder: Kjell Inge Tomren
Mai 2021

Bacheloroppgave

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

NTNU
Kunnskap for en bedre verden

Lasse Kregnes Hansen, 10009
Pål Nyseth, 10093
Martin Håhjem Årdal, 10113

Informasjonsskjerm for sjakkturneringer

Bacheloroppgave i Dataingeniør
Veileder: Kjell Inge Tomren
Mai 2021

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

NTNU
Kunnskap for en bedre verden

FORORD

Om Oss

Gruppen består av tre dataingeniør-studenter fra NTNU i Ålesund. Da gruppen skulle starte å søke på bacheloroppgaver, var alle lystne på å oppgaver som involverte fullstack utvikling, noe hele gruppen ble kjent med gjennom faget Mobile og Distribuerte Applikasjoner. Alle våre valg innebar fullstack-utvikling, hvor vi hadde denne oppgaven som førstevalg. Denne oppgaven har gitt oss flere erfaringer og kjennskaper til teknologier som er nyttige for arbeidslivet.

Om Oppdragsgiver

Oppdragsgiver for bacheloroppgaven var Aalesund Schaklag. Aalesund Schaklag ble stiftet i 1913. De har lokaler sentralt i Ålesund by. De arrangerer sjakkturneringer for både barn og voksne, samt treninger for barn og ungdom. De har, under normale omstendigheter, åpne lokaler en dag i uken hvor medlemmer får samles.

Aalesund Schaklag var representert av deres nestleder og CTO hos Driw AS Arne Unneland.
Med sin nære kjennskap til både sjakk og lang erfaring innenfor programvareutviklingsbransjen var Arne Unneland en sterk ressurs for gruppen, hvor han kunne forklare sjakk-konsepter, samt gi oss råd og tips ved utvikling, teknologier og arkitektur.

Andre bidragsytere

Vi vil også rette en takk til vår veileder, Kjell Inge Tomren. Kjell Inge Tomren har gitt god veiledning gjennom hele prosjektet og oppmuntring når det trengtes.

SAMMENDRAG

I dag bruker oppdragsgiver, Aalesund Schaklag, hovedsakelig to programvarer for turneringer i lokaler. De bruker Turneringsservice for å organisere turneringer og DGT LiveChess for å vise sjakkpartier, som foregår på digitale sjakkbrett.

Disse programvarene gir Aalesund Schaklag, publikum og deltagere for deres turneringer noen utfordringer. Informasjonen rundt turneringene er spredt over to tjenester som ikke snakker med hverandre, samt det kan ta betraktelig tid før denne informasjonen er tilgjengelig for publikum og deltagere.

Det manglet altså en samling og strukturering av informasjonen fra disse to programvarene. Vår løsning hadde som mål å samle informasjonen fra disse to tjenestene og presentere det på en felles skjerm i kort tid etter informasjonen var tilgjengelig.

Løsningen ble en tjeneste med en back-end med en Spring-Boot-applikasjon og en PostgreSQL-database og en front-end bestående av JavaScript og JavaScript-rammeverket Vue.js.

Resultatet er system som består av en fungerende back-end som kjører en serverapplikasjon som kommuniserer med en database, og en fungerende front-end som kommuniserer med back-end. Systemet oppfyller mange av kravene.

Rapporten vil gi en oversikt over hvordan oppgaven ble løst.

TERMINOLOGI

Forkortelser

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CLI	Command Line Interface
CSR	Client-Side Rendering
CSS	Cascading Style Sheet
CTO	Chief Technology Officer
DGT	Digital Game Technology
DI	Dependency Injection
DOM	Document Object Model
FEN	Forsyth-Edwards Notation
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
IoC	Inversion of Control
JSON	JavaScript Object Notation
JWT	JSON Web Token
PGN	Portable Game Notation
REST	Representational State Transfer
SFC	Single File Components
SSH	Secure Shell Protocol
SoC	Separation of Concerns
SPA	Single Page Application
SSR	Server Side Rendering
TSL	Transport Security Layer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

INNHOLDSFORTEGNELSE

FORORD I

SAMMENDRAG II

TERMINOLOGI III

INNHOLDSFORTEGNELSE IV

FIGURLISTE VII

1 INTRODUKSJON 1

1.1 RAPPORTSTRUKTUR 1
1.2 BAKGRUNN 1
1.3 PROBLEMSTILLING 1
1.4 MÅLSETTING 2

2 TEORETISK GRUNNLAG 3

2.1 GIT FLOW 3
2.2 SMIDIG UTVIKLINGSMETODIKK 3
2.2.1 Scrum 4
2.2.2 Scrum-roller 4
2.2.3 Scrum-artefakter 4
2.2.4 Scrum-møter 5
2.2.5 Sprint-Rapport 5
2.2.6 Kanban-brett 5
2.2.7 User Story 6
2.2.8 Epics 6
2.2.9 Estimering 6
2.3 DESIGN OG ARKITEKTUR 6
2.3.1 Client Side Rendering 6
2.3.2 Single Page Application 6
2.3.3 Separation of Concerns 7
2.3.4 Komponentbasert arkitektur 7
2.3.5 Lagvis-arkitektur 7
2.3.6 REST 7
2.3.7 MVC 8
2.4 SPRING KONSEPTER OG ARKITEKTURER 9
2.4.1 Dependency Injection 9
2.4.2 Inversion of Control 10
2.4.3 IoC Container 10
2.4.4 Bean 10
2.5 VUE KONSEPTER 10
2.5.1 Reaktivitet og databinding 10
2.5.2 Komponenter 10
2.5.3 Lasting av komponenter 11
2.5.4 Template Syntax 11
2.5.5 Single File Components 11
2.5.6 Komponentlivssyklus 11
2.5.7 Client Side Routing 12
2.6 SIKKERHET 12
2.6.1 JWT 12
2.6.2 HTTPS 12
2.7 DATABASER 12
2.7.1 Relasjonsdatabase 12
2.7.2 Primærnøkkel 13
2.7.3 Sekundærnøkkel 13
2.7.4 Et-til-et forhold 13

2.7.5 *Et-til-mange forhold* ... 13
2.7.6 *Mange-til-mange forhold* ... 13
2.7.7 *SQL* ... 13
2.7.8 *ORM* ... 13
2.8 SJAKKNOTASJONER OG -FILFORMATER ... 14
2.8.1 *Portable Game Notation* ... 14
2.8.2 *Forsyth-Edwards Notation* ... 14
2.8.3 *TRX* ... 14
2.9 SPRÅK, RAMMEVERK OG TEKNOLOGIER ... 15
2.9.1 *Java* ... 15
2.9.2 *JavaScript* ... 15
2.9.3 *HTML* ... 15
2.9.4 *CSS* ... 15
2.9.5 *Rammeverker og teknologier for Java* ... 15
2.9.6 *Rammeverker og biblioteker for JavaScript og CSS* ... 17
2.9.7 *Verktøy, rammeverker og biblioteker for Vue.Js* ... 18

3 MATERIALER OG METODE ... 19

3.1 ORGANISERING AV PROSJEKT ... 19
3.1.1 *Prosjekt Team* ... 19
3.1.2 *Møter med veileder* ... 19
3.1.3 *Møter med oppdragsgiver* ... 19
3.1.4 *Møter innad i gruppen* ... 19
3.2 SMIDIG UTVIKLINGSMETODIKK ... 19
3.2.1 *Sprint planleggingsmøte* ... 20
3.2.2 *Sprint gjennomgangsmøte* ... 20
3.2.3 *Sprint Retrospektivmøte* ... 20
3.2.4 *Daglig Scrum-møte* ... 20
3.3 VERKTØY FOR PROSJEKTSTYRING ... 20
3.3.1 *Jira* ... 20
3.3.2 *Confluence* ... 20
3.3.3 *Discord* ... 21
3.3.4 *Git Flow* ... 21
3.4 PLANLEGGING ... 21
3.4.1 *Forprosjektsrapport* ... 21
3.4.2 *Informasjonsinnsamling* ... 21
3.4.3 *Wireframes* ... 21
3.4.4 *Kravspesifikasjoner* ... 22
3.5 TEKNOLOGIER FOR UTVIKLING ... 22
3.5.1 *Presentasjonslaget - Frontend* ... 22
3.5.2 *Datalaget - Backend* ... 24
3.5.3 *Andre verktøy og teknologier* ... 25

4 RESULTATER ... 27

4.1 FUNKSJONALITET FOR SLUTTBRUKER ... 27
4.1.1 *Visning av turneringer* ... 27
4.1.2 *Turneringsskjerm* ... 28
4.1.2.3 *Visning av en kamp* ... 29
4.1.3 *Roller i systemet* ... 31
4.1.4 *Registrering* ... 31
4.1.5 *Login* ... 32
4.1.6 *Opprette turneringer* ... 32
4.1.7 *Dashboard* ... 33
4.1.8 *Sende meldinger* ... 34
4.1.9 *Slette meldinger* ... 35
4.1.10 *Slette turnering* ... 35
4.1.11 *Profil* ... 35
4.1.12 *Administrator-panel* ... 36

4.2 BACK-END ... 37
4.2.1 Lagvis arkitektur ... *37*
4.2.2 REST ... *38*
4.2.3 Detaljert gjennomgang av implementering av lagvis arkitektur ... *39*
4.2.3.1 Eksempel – Meldingstjeneste ... *40*
4.3 FRONTEND ... 44
4.3.1 Single Page Application ... *44*
4.3.2 Client Side Routing ... *44*
4.3.3 Single File Components ... *45*
4.3.4 Vuex og tilstandsbehandling ... *46*
4.3.5 Inputvalidering ... *46*
4.3.6 Axios - HTTP klient ... *46*
4.3.7 Utseende ... *47*
4.4 DATABASE ... 48
4.4.1 Design ... *48*
4.4.2 Tabeller ... *49*
4.4.3 Forhold ... *51*
4.4.4 Befolke med testdata ... *52*
4.5 SIKKERHET ... 53
4.5.1 HTTPS ... *53*
4.5.2 JWT ... *54*
4.6 DETALJERT GJENNOMGANG AV IMPLEMENTASJON AV BRUKERTJENESTER. ... 54
4.6.1 Brukertjenester - Backend ... *54*
4.6.2 Brukertjenester - Frontend ... *59*
5 DRØFTING ... 83
5.1 TEKNISK RESULTAT ... 83
5.1.1 Hele løsningen ... *83*
5.1.2 Frontend ... *83*
5.1.3 Back-end ... *84*
5.1.4 Database ... *84*
5.2 GJENNOMFØRELSE AV PROSJEKTET ... 85
5.2.1 Organisering ... *85*
5.2.2 Prosjektstyring ... *85*
5.2.3 Scrum ... *85*
5.3 Planlegging og tidsbruk ... *85*
5.3.1 PERIODE MED INFORMASJONSINNSAMLING ... 86
5.3.2 UNDERESTIMERING AV TID ... 86
5.3.3 SEN START AV UTVIKLING ... 86
5.4 FORBEDRINGER OG UTVIDELSER ... 87
5.4.1 Manglende konsepter ... *87*
5.4.2 Bruke Sockets for turneringer ... *88*
5.4.3 Implementere et faktisk REST-API ... *88*
5.4.4 Ny arkitekturer for løsningen ... *88*
6 KONKLUSJON ... 89
6.1 TEKNISK RESULTAT ... 89
6.2 GJENNOMFØRELSE AV OPPGAVEN ... 89
7 REFERANSER ... 91

FIGURLISTE

Figur 1 - Illustrasjon av MVC-konseptet 9
Figur 2 - Eksempel av en PGN - Viser et ferdigspilt parti 14
Figur 3 - Eksempel av en FEN-streng - Viser startposisjon på sjakkbrett 14
Figur 4 - Visning av turneringer 27
Figur 5 - Visning av turneringsskjerm 28
Figur 6 - Visning av ferdigspilt parti 29
Figur 7 - Eksempel av turnering med et parti 29
Figur 8 - Eksempel av turnering med flere partier 30
Figur 9 - Eksempel av visning av meldinger sendt til turnering. 30
Figur 10 - Presentasjon av viktig melding 30
Figur 11 - Presentasjon av ikke viktig melding 30
Figur 12 - Registreringsskjema. 31
Figur 13 - Innloggingsskjema 32
Figur 14 - Knapp for oppretting av turnering 32
Figur 15 - Skjema for oppretting av turnering 33
Figur 16 - Visning av Dashboard for turnering 34
Figur 17 - Eksempel av deaktivert send-knapp 34
Figur 18 - Eksempel av aktivert send-knapp. 34
Figur 19 - Slett-knapp plassert til høgre i melding. 35
Figur 20 - Slett-knapp for turnering plassert nederst i Dashboard. 35
Figur 21 - Visning av brukerprofil 36
Figur 22 - Visning av Administrator-panel 36
Figur 23 - Brukerdetaljer - andre brukere 37
Figur 24 - Brukerdetaljer - Administrator 37
Figur 25 - Illustrasjon av lagvis arkitektur implementert med Spring annoteringsnavn. .37
Figur 26 - Informasjonsflyt for meldingstjeneste 40
Figur 27 - Eksempel av at man legger til JWT i header til en forespørsel 40
Figur 28 - Eksempel av meldingsobjekt i JSON-form vist i Postman 41
Figur 29 - Implementasjon av sendMessage-metoden i MessageController 41
Figur 30 - Implementasjon av sendMessage-metoden i MessageService 42
Figur 31 - Implementasjon av MessageRepository 42
Figur 32 - Illustrasjon av lagvis arkitektur 43
Figur 33 - Implementasjon av getMessageTournament-metode i MessageController 43
Figur 34 - Implementasjon av getMessageTournament-metode i MessageService 43
Figur 35 - Innholdet i bygg-mappen. Viser bare en HTML-fil 44
Figur 36 - Eksempel av definisjonen av en rute. 45
Figur 37 - Eksempel av Router-link rundt HTML-element for navigering 45
Figur 38 - Eksempel av strukturen av en Single File Component 45
Figur 39 - Eksempel av bruk av Vuelidate funksjon 46
Figur 40 - Eksempel av referanse til Vuelidate-property i Template Syntax 46
Figur 41 - Eksempel av definering av Vuelidate regler. 46
Figur 42 - Eksempel av oppretting av Axios-instans 47
Figur 43 - Eksempel av en funksjon i en service-fil. 47
Figur 44 - Eksempel av syntax for håndtering av respons ved asynkrone kall. 47
Figur 45 - Diagram for databasedesign 49
Figur 46 - Eksempel av definisjon av et mange-til-mange forhold 51
Figur 47 - Eksempel av definisjon av et et-til-mange-forhold 52
Figur 48 - Eksempel av ene siden av et tosidig et-til-et-forhold 52
Figur 49 - Eksempel av den andre siden av et tosidig et-til-et-forhold 52
Figur 50 - Eksempel av nødvendig konfigurasjon for HTTPS i Spring-rammeverket. 53
Figur 51 - Eksempel av kode som kreves for at HTTPS skal bli aktivert ved oppstart av server. 53
Figur 52 – Eksempel av plassering av sertifikat. 54

Figur 53 - Informasjonsflyt for autorisering ved login 55
Figur 54 - Eksempel av bygging av JWT 56
Figur 55 - Informasjonsflyt for autorisering for andre ressurser som krever autorisering 57
Figur 56 - Implementasjon av metode for å endre en brukers rolle. 58
Figur 57 - Implementasjon av changeRole-metoden, med SQL-query 59
Figur 58 - Registreringsskjema 59
Figur 59 - Eksempel av felt som ikke har gyldig input, her inget brukernavn fylt inn 60
Figur 60 - Eksempel av felt som ikke har gyldig input, for kort brukernavn 61
Figur 61 - Implementasjon av sendUserData-funksjon som henter ut validert data fra Vuelidate..61
Figur 62 - Utsnitt av kode - Om data i kompomponent ikke inneholder feil, så kan man gjennomføre kall. 61
Figur 63 - Implementasjon av register-funksjonen i service-klassen GameService. 62
Figur 64 - Innloggingsskjema 62
Figur 65 - Eksempel av bruk av mapActions 63
Figur 66 - Implementasjon av asynkron login-funksjon i Vuex modul. 63
Figur 67 - Implementasjon av asynkron login-funksjon i Login-komponent 64
Figur 68 - Eksempel av hvordan man lagrer til Vuex 64
Figur 69 - Implementasjon av saveTokenData som lagrer JWT og informasjon det inneholder. 64
Figur 70 - Implementasjon av funksjonen isTokenActive 65
Figur 71 - Implementasjon av getLoginStatus-funksjon og mapping av funksjoner fra Vuex store til komponent 66
Figur 72 - Eksempel av dynamisk innlasting av Router-link. 66
Figur 73 - Eksempel av definisjon av rute hvor det kreves autorisering 67
Figur 74 - Eksempel av bruk av beforeEach-funksjon for ruting, del 1 67
Figur 75 - Eksempel av bruk av beforeEach-funksjon for ruting, del 2 67
Figur 76 - Eksempel av implementasjon av getters-funksjon 68
Figur 77 - Eksempel av mapping av funksjoner fra Vuex 68
Figur 78 - Eksempel av dynamisk innlasting av elementer 68
Figur 79 - Visning av profil med for andre brukere 69
Figur 80 - Visning av profil for Administrator 69
Figur 81 - Implementasjon av getTournamentsByOwner-funksjon. 69
Figur 82 - Eksempel av dynamisk innlastning elementer basert på bruker har turneringer og er enten Administrator eller arrangør 70
Figur 83 - Eksempel på visning av egne turneringer. 70
Figur 84 - Eksempel på oppretting av Axios instans 70
Figur 85 - Konfigurasjon av Axios instans med interceptor 71
Figur 86 - Implementasjon av generell get-funksjone for autorisering. 71
Figur 87 - Visning av opprettning av turnering. 72
Figur 88 - Skjema for oppretting av turnering 72
Figur 89 - Visning av en dato-velger. 73
Figur 90 - Kode som lagrer validering input for en turnering. 73
Figur 91 - Implementasjon av post-funksjon for opprettelse turnering 74
Figur 92 - Eksempel av Dashboard for en turnering 74
Figur 93 - Eksempel av betinget visning av element 75
Figur 94 - Implementasjon av ownerOfTournament-funksjon 75
Figur 95 - Implementasjon av adminiOrOwner-funksjon. 75
Figur 96 - Eksempel på visning av beskjed gitt i Dashboard til bruker dersom om brukeren ikke er eier av turneringen 75
Figur 97 - Implementasjon av fetchMessage-funksjon 76
Figur 98 - Visning av sendte meldinger i Dashboard. 76
Figur 99 - Eksempel på elementet som viser meldingene. 76

Figur 100 - Implementasjon av setImportance som bestemmer om en melding er viktig eller ikke i frontend. 77
Figur 101 - Implementasjon av slette-knapp for meldinger....77
Figur 102 - Implementasjon av deleteMsg-funksjon....77
Figur 103 - Skjema for sending av meldinger....78
Figur 104 - Visning av valg av viktighet av melding. 78
Figur 105 - Eksempel på validering av melding. 78
Figur 106 - Implementasjon av sendMessage-funksjon, del 1 79
Figur 107 - Implementasjon av sendMessage-funksjon, del 2 79
Figur 108 - Implementasjon av deleteTournament 80
Figur 109 - Skjema for endring av en brukers rolle....80
Figur 110 – Eksempel på betinget lasting av element....80
Figur 111 - Eksempel på validering i felt for bruker-id....81
Figur 112 - Eksempel på valg av ny rolle til bruker 81
Figur 113 - Visning av mapping mellom verdier og tekst gjennom et option-element 81
Figur 114 - Implementasjon av updateRole-funksjonen, del 1....82
Figur 115 - Implementasjon av updateRole-funksjonen, del 2....82
Figur 116 - Implementasjon av updateRole-funksjonen, del 3....82

1 INTRODUKSJON

I dette kapittelet vil det bli oppgitt hva som var bakgrunnen og problemstillingen for oppgaven, hvilke utfordringer løsningen skal løse og mål med løsningen.

1.1 Rapportstruktur

Dette delkapittelet skal vise strukturen av rapporten.

Kapittel 1 - Introduksjon
Vil gi en innledning for oppgaven, med bakgrunnsinformasjon, problemstilling og mål med løsningen.

Kapittel 2 - Teoretisk grunnlag
En oversikt over teori og kunnskap som er tatt i bruk i løsningen av oppgaven.

Kapittel 3 - Materialer og metoder
En oversikt over materialer og metoder som er tatt i bruk i løsningen av oppgaven.

Kapittel 4 - Resultat
Beskriver de oppnådde resultatene i oppgaven.

Kapittel 5 - Diskusjon
Vil gi en diskusjon av resultater og materialer og metoder som ble brukt i løsningen av oppgaven.

Kapittel 6 - Konklusjon
En kort gjengivelse av punkter diskutert i resultat og diskusjon, som trekker en konklusjon over utført arbeid.

1.2 Bakgrunn

Aalesund Schaklag er sjakklubb basert i Ålesund By, som lar sjakkinteresserte og aktive sjakkspillere samles for treninger og sjakkturneringer. I sjakkturneringer de arrangerer, bruker de to applikasjoner i gjennomførelsen av turneringene. Applikasjonene er Turneringsservice og DGT LiveChess. Turneringsservice er et verktøy for administrasjon av turneringer. DGT LiveChess er verktøy for visning og kringkasting av sjakkpartier i sanntid. DGT LiveChess henter informasjonen fra digitale sjakkbrett.

1.3 Problemstilling

De to nevnte applikasjonene, Turneringsservice og DGT LiveChess gir Aalesund Schaklag, publikum og deltagere for deres turneringer noen utfordringer. Informasjonen rundt turneringene er spredt over to tjenester som ikke snakker med hverandre, samt

det kan ta betraktelig tid før noe av denne informasjonen er tilgjengelig for publikum og deltagere.
Det manglet altså en samling og strukturering av informasjonen fra disse to programvarene. Vår løsning hadde som mål å samle informasjonen fra disse to tjenestene og presentere det på en felles skjerm i kort tid etter informasjonen var tilgjengelig, samt gi turneringsledere muligheter til å dele beskjeder med publikum.

1.4 Målsetting

Mål for oppgaven er

1) Å utvikle en nettside som en Single Page Application
    a) hvor publikum og deltagere ved turnering kan følge en turnering.
    b) hvor ledelsen i sjakkturneringene kan vise
        i) sjakkpartier fra digitale sjakkbrett
        ii) resultatlister
        iii) rundeoversikter
        iv) meldinger
2) Å utvikle en REST-API med database for å samle nødvendig informasjon å gjøre det mulig presentere det i en turneringskontekst.
3) Hente informasjon fra PC-er hos turneringsledelse og sende den til backend gjennom REST-API.

Hva løsningen ikke skal være

1) En tjeneste som erstatter eksisterende programvare som blir brukt ved sjakkturneringer i dag.

2 TEORETISK GRUNNLAG

I dette kapittelet gis det en oversikt over teori og kunnskap som er tatt i bruk i løsningen av oppgaven.

2.1 Git Flow

Git Flow er en metode for å strukturere sin arbeidsprosess ved bruk av versjonskontrollsystemer, slik som Git. Git Flow definerer 5 typer grener, Develop, Main, Feature, Release og Hotfix. Main-grenen skal holde alle offisielle utgivelser av en programvare. Develop-grenen er hvor nye funksjonaliteter under utvikling hører til. En Feature-gren tar utgangspunkt i Develop-grenen, og er de grenene hvor ny funksjonalitet utvikles. Når en funksjonalitet er ferdig utviklet, slås denne grenen sammen med Develop-grenen. Release-grenen skal inneholde all kode som skal inngå i en nyutgivelse av programvaren. Ingen ny utvikling skal skje på en slik gren, bare arbeid knyttet til utgivelsen. Release-grenen tar utgangspunkt i Develop-grenen. Når man er klare for en utgivelse, slår man sammen denne Release-grenen med Main-grenen. Hotfix-grenen er hvor fikser av den utgitte programvaren skal ta sted. Hotfix-grenen tar utgangspunkt i Main-grenen og når man har utbedret feilen, slås Hotfix-grenen sammen med Main-grenen.
Metoden er ofte brukt i prosjekter med sjeldne utgivelser. [1]

2.2 Smidig Utviklingsmetodikk

Innenfor Smidig utviklingsmetodikk, også kalt Agile, vil man finne frem til kravspesifikasjon for programvaren og utvikle den gjennom hyppig samarbeid og kommunikasjon med kunder. Utviklerne organisere seg selv i små kryssfunksjonelle teams. Man deler arbeidsoppgavene opp i mindre oppgaver hvor de kan jobbes på i korte arbeidsperioder. I hver periode planlegger, designer, implementerer og tester man programvaren. Dette vil bety at man minimerer behovet design og planlegging tidlig i prosessen, da dette gjøres for hver del i hver arbeidsperiode. Målet for hver arbeidsperiode er å levere verdi til kunden gjennom en fungerende programvare og ny funksjonalitet. Hyppig kommunikasjon mot kunde gjør at man avdekker eventuelle feilspesifiseringer av programvaren og endrede eller nye behov fra kunden. [2]
"The agile manifesto" eller, manifestet for smidig programvareutvikling, sier noe om prioriteringene ved bruk av denne utviklingsmetoden:

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

Punktene til høyre har fortsatt verdi, men punktene til venstre som blir verdsatt enda høyere.

Metodikken kom som et motsvar mot fossefallsmodellen. Forenklet kan man si at fossefallsmodellen består av 5 steg, kravbeskrivelse, design, implementasjon, testing og vedlikehold av en programvare. Hvert steg omhandlet hele programvaren, og man startet ikke på neste steg før man ferdig med det første. Modellen har ikke fokus på

kommunikasjon mot kunder. Dette kunne føre til at man produserte programvare som ikke passet kunden sine behov da man gjorde en feil tidlig i prosessen. [3] [4]

2.2.1 Scrum

Scrum er et rammeverk adopterer prinsippene fra smidig utviklingsmetodikk. Innenfor Scrum jobber man i korte arbeidsperioder, kalt Sprinter. Sprintene går gjerne over 1-4 uker. Målet for hver sprint er, som i Smidig utviklingsmetodikk, å levere håndfast verdi til kundene. Scrum definerer også roller, artefakter og møter for effektiv styring av programvareutvikling i utviklingslag. [5]

2.2.2 Scrum-roller

Innenfor Scrum finnes det tre roller. Produkteier, Scrum utviklingslag og Scrum-mester. Alle disse tre rollene bekles av personer som er direkte ansvarlig for å utvikle produktet, ikke av personer utenfor denne gruppen.

Produkteier er ansvarlig for at brukerne får verdi fra programvaren. Dette involvere å bestemme hvilke brukerhistorier (beskrive i 2.1.6) fra produktbacklogen (beskrive i 2.1.8) som skal prioriteres. Ingen andre enn produkteier har rettigheter til å endre rekkefølgen i backlogger. Produkteier kan fortsatt være en som utvikler programvaren.

Scrum utviklingslaget er en selvorganisert gruppe som ansvarlig for å utvikle programvaren. Denne gruppen planlegger og estimerer, beskrevet i 2.1.9, brukerhistoriene fra produkt-backloggen. De tar bare brukerhistorier de rekker i en Sprint, og de må ta brukerhistorier fra toppen av produkt-backloggen.
Scrum-mester er en tilrettelegger for Scrum. Scrum-mester skal se til at Scrum-metodikken og den prinsipp blir fulgt. I praksis kan det blant annet bety at Scrum-mesteren utfører oppgaver fra det å kalle inn til de ulike Scrum-møtene (beskrevet i 2.1.5), til å hjelpe produkteier i å prioritere i produkt-backloggen. [6]

2.2.3 Scrum-artefakter

Scrum-artefakter er verktøy og informasjon som er brukt til å beskrive produktet og gi forståelse for produktet som blir utviklet, planlagte kommende aktiviteter og aktiviteter som allerede er utført.

Dette uttrykkes hovedsakelig gjennom to artefakter, Produkt-backlog og Sprint-backlog

Produkt-backloggen er en prioritert liste over brukerhistorier, funksjoner og feilrettinger. Den gir alle interessenter en oversikt over funksjonaliteter som produktet som blir utviklet skal ha. Det er bare Produkteier som kan endre prioriteten i produkt-backloggen. Det som Scrum utviklingslaget har kommet fram til skal jobbes med, flyttes til Sprint-backloggen under Sprint-planlegging (beskrevet i 2.1.5).

Sprint-backloggen består av det arbeidet som Scrum utviklingslaget har bestemt skal utføres i kommende Sprint. Ut ifra denne backloggen deler Scrum utviklingslaget arbeidet opp i mindre, mer konkretiserte Sprint-oppgaver, som beskriver hvordan arbeidet skal løses. [7] [8]

2.2.4 Scrum-møter

Scrum-møter, også kalt seremonier, finnes i fire former. De er Sprint-planleggingsmøte, Daglig Scrum-møte, Sprint gjennomgangsmøte, og Sprint retrospektivmøte.

Sprint-planleggingsmøtet er det første møtet i en Sprint. Dette møtet er hvor det bestemmes hva som skal jobbes med i den nye Sprinten. Produkteier beskriver et overordnede mål for Sprinten og hvilke produkt-backlog punkter som oppfyller dette målet. Scrum utviklingslaget, sammen med Produkteier, diskuterer hvilke punkter er gjennomførbare og skal tas med inn i Sprinten og over til Sprint-backloggen. Det gjøres med at Produkt-backlog punktene estimeres (beskrevet i 2.1.9) og deles opp i mindre Sprint-oppgaver.

Det daglige Scrum-møtet, også kalt Stand Up, er et kort, daglig møte hvor alle rollene er til stede. Alle deltakere snakker kort og uformelt om hva man jobbet med forrige arbeidsdag, hva man skal jobbe med i dag, og om det er noe som hindrer man i å utføre oppgavene. Dette møtet har som hensikt å gi hele gruppen en oversikt over fremgang og avdekke hindringer i fremgangen.

Sprint-gjennomgangsmøtet blir gjennomført siste dagen av en Sprint. I dette møtet presenterer Scrum-utviklingslaget en fungerende versjon av applikasjonen for Produkteier og kunden. Både Produkteier og kunden gir tilbakemelding på utført arbeid. Kunden kan velge å godta eller avslå arbeidet, samt komme med nye forslag til videre utvikling.

Sprint retrospektivmøte er det aller siste møtet for en Sprint. Det er ment som et møte for alle personene i Scrum-teamet, hvor man fokuserer på hva på hvordan teamet fungerer, hva de har gjort bra og hva som kan forbedres i hensyn til effektiviteten til teamet. [9]

2.2.5 Sprint-Rapport

En Sprint Rapport er et sammendrag over fremgangen for en Sprint. Den viser i vanlig alle oppgaver knyttet til en Sprint, deres estimerte verdi, om de er utført eller ikke og en visualisering av fremgangen. [10]

2.2.6 Kanban-brett

Kanban er en Lean-utviklingsmetodikk. [11] Innenfor Kanban blir arbeidsoppgaver visualisert på et brett, med ulike kolonner. Hver kolonne representerer en tilstand eller status en arbeidsoppgave kan ha. Tilstanden er arrangert fra starttilstand på venstre side til sluttilstand på høgre side. Dette lar man enkelt se fremgangen for en enkelt arbeidsoppgave. I flere digitale implementasjoner av Scrum-metodikken brukes et slikt Kanban-brett. Innenfor Scrum har vanligvis arbeidsoppgavene ha statusene ToDo, Påbegynt og Ferdig. [12]

2.2.7 User Story

I Scrum definerer man krav og funksjonalitet et produkt skal ha gjennom brukerhistorier, kalt User Stories innen Scrum. En brukerhistorie gis på formatet "As a user… I want to… So that". Brukerhistorien beskriver hvilken bruker i systemet har et ønske, hva ønske er og hvorfor brukeren har ønsket. De skal altså beskrive hvordan programvaren kan gi en verdi for en bruker. Legges til i produkt-backlogger og sprint-backlogger. Er ikke et Scrum-eksklusivt konsept. [13]

2.2.8 Epics

Større historier kan beskrives som Eposer, også kalt Epics. Disse større historiene er historier som er for store til å leveres i en sprint-iterasjon. Eposer lar man gruppere brukerhistorier til et større arbeid med et felles mål. [14]

2.2.9 Estimering

Innenfor Smidig Utviklingsmetodikk estimerer man omfanget av arbeidsoppgavene ved hjelp av Story Points. Innenfor Scrum gjøres dette i Sprint-planleggingsmøter. Estimering er en gruppeaktivitet med hele utviklingsteamet involvert. I starten av utviklingen kommer teamet frem til en konsensus for hvor mye arbeid et Story Point innebærer. Så gir man arbeidsoppgave så mange Story Points man mener den fortjener. Teamet må komme frem til hvor mange Story Points sammen, gjerne gjennom en samtale over kompleksitet og utfordringer for oppgaven. [15]

2.3 Design og Arkitektur

Dette delkapittelet beskriver design, designprinsipper og arkitekturkonsepter relevante for denne oppgaven.

2.3.1 Client Side Rendering

Client-Side-Rendering, CSR, lar nettsider bli gjengitt i sin helhet med bruk av JavaScript. Ved bruk av CSR endrer man html-elementer, innhold og utseende ved å direkte interaktere med en nettleseres Document Object Model, DOM, -API. Dette innebærer at en klient, for eksempel en nettapplikasjon, må stå for ruting til innhold, henting av data og inneholde mye mer logikk. Er et konsept sterkt knyttet til Single Page Applications, SPA. Motsetningen til CSR er Server Side Rendering, SSR hvor man ruter til en URL og får returnert en ferdig strukturert HTML-fil med alle nødvendig html-elementer, utseende og logikk bakt inn. [16]

2.3.2 Single Page Application

En Single Page Application, SPA, er nettapplikasjon eller nettside hvor man dynamisk omskriver innholdet på en side. I en SPA lever alt innhold på en side, gjerne index.html. Et viktig moment med Single Page Applications er at siden ikke blir oppdatert, siden man ikke bytter side. I stedet blir innholdet på siden byttet ut. Det skaper en mer sømløs opplevelse for brukeren, samt man får holde på tilstanden på siden. Dette krever at nettapplikasjonen må inneholde logikk for endring av innhold og utseende, samt kunne hente nødvendig data og kunne sende brukere til riktig innhold. [17]

2.3.3 Separation of Concerns

Separation of Concerns, SoC, er et design prinsipp hvor man deler strukturen av koden inn i separate deler hvor hver del har klart definerte roller og ansvar. I det enkleste, så vil man unngå at samling av kode har mer enn et ansvar. [18] [19]

2.3.4 Komponentbasert arkitektur

Moderne JavaScript-rammeverker følger en komponentbasert arkitektur. Kode blir samlet i løst knyttet og gjenbrukbare komponenter som kan komponeres sammen med andre komponenter. I noen rammeverk vil også CSS og HTML bli inkludert i samme komponent gjennom rammeverkstøtte. Tanken er da at man oppnår SoC ved at en komponent inneholder alt den trenger, og ikke noe mer, for å presenteres til bruker, selv om den inneholder bare kode. [20]

2.3.5 Lagvis-arkitektur

Lagvis-arkitektur, er et design hvor man deler en arkitektur inn i lagvis deler. Arkitekturen vil normalt fremstå med fire lag, hvor hvert lag har ulike roller og ansvar. Disse lagene er Presentasjonslaget, Forettningslogikklaget, Persistenslaget og Databaselaget. Presentasjonslaget har ansvar for å presentere data til bruker. Et REST-API kan være Presentasjonslaget. Presentasjonslaget spør om og gir data til Forettningslogiklaget. Forettningslogiklaget behandler data og utfører logikk mot dataen og sender dataen til enten Presentasjonlaget eller Persistenslaget. Persistenslaget har ansvar for å hente fra eller lagre data til databaselaget. Database-laget er laget hvor dataen blir persistert. Denne arkitekturen gir høy SoC, da hvert lag har klart separerte roller og ansvar. [21] [22]

2.3.6 REST

Representational State Transfer, REST, er en arkitekturstil som bygger på HTTP. Ved at man bruker HTTP som protokoll, lar man oppnå tilstandsløs kommunikasjon.

REST spesifiserer at nettressurser skal gjøres tilgjengelig gjennom en tekstlig representasjon i form av en URI. En nettressurs kan være alle typer objekter eller handlinger som kan identifiseres, navngitt, adressert, behandlet eller utført på nettet. For eksempel kan en ressurs for en informasjon om kommunen Sykkylven gjøres tilgjengelig på formen "/api/kommune/sykkylven". De mest brukte filformatene for å sende informasjon i REST-systemer er XML og JSON.

REST tar i bruk bare de HTTP-metodene man trenger for å behandle data og ressurser. De metoden er GET, POST, PUT og DELETE.
REST har som mål om, blant annet, å gi systemer høy ytelse, skalering og enkle grensesnitt, gjennom bruken av HTTP som en tilstandsløs protokoll og standard operasjoner og metoder.

For at et system skal være kunne si at det følger REST, så må det følge seks begrensninger.

1) Klient - Server arkitektur

Man separerer brukergrensesnitt fra lagring av data. Gjennom dette oppnår man SoC, i og med server har ansvar for data, klienten har ansvar for presentasjon av dataen, men begge kan eksistere uten den andre.

2) Tilstandsløshet

Kommunikasjon skal være tilstandsløs, som vil si at man ikke beholder noe sesjonsdata hos mottaker og hver respons som blir sendt kan bli forstått i isolasjon. Dette oppnår man gjennom bruk av HTTP.

3) Støtte for Caching

Både klienter og servere skal kunne cache responser. Responsene selv må derav kunne holde på informasjon om de kan caches eller ikke, så man unngår at klienter holder på en eldre tilstand av dataen. Caching er viktig for å kunne gi høy ytelse.

4) Lagdelt system

Klient og server skal kunne eksistere i et lagdelt system. Man skal kunne introdusere andre lag mellom klient og server, slik som en proxy eller reverse proxy, et sikkerhetslag foran server, osv. Klient og server skal fortsatt kunne kommunisere sammen når disse introduseres.

5) Uniformt grensesnitt

Systemet skal ha et uniformt grensesnitt, hvor systemet skal støtte de følgende begrensningene.

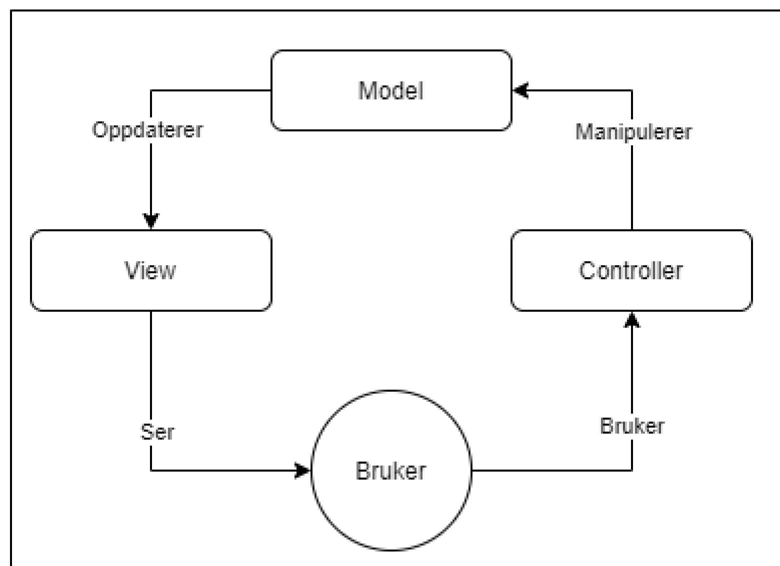
a) Ressurser er identifiserbare gjennom en forespørsel
b) Klient skal kunne manipulere en ressurs om det holder på en representasjon av ressursen
c) Hver beskjed skal inneholde nok informasjon til å beskrive hvordan man behandler beskjeden, oftest gjort gjennom spesifisering av media/applikasjonstype.
d) Server skal kunne gi klienter hypertekstlinker til andre relevante ressurser som server har tilgjengelig. Dette gjør det mulig for klienter å dynamisk respondere på dataen som serveren sender, og man unngår hardkoding på klientsiden for hvor man finner ressurser.

6) Kode på kommando.

Server kan ha funksjonaliteten til å sende kode til klienter, f.eks. JavaScript-kode. Kode på kommando er frivillig å implementere. [23] [24]

2.3.7 MVC

MVC står for Model View Controller og er en ofte brukt modell innenfor programmering. Hovedessensen av MVC er å lukke mellomrommet mellom bruker og datamaskin.



Figur 1 - Illustrasjon av MVC-konseptet

En MVC er bygd opp av tre deler, en modell, et view og en kontroller, her vist i Figur 1.

Modell - Modeller representerer kunnskap og kan bli sett på som et eller flere objekter av ekte ting. På den ene siden har man et 1→1 forhold mellom modell og dens deler, og på den andre siden er hvordan verden blir oppfattet av eieren av modellen.
View - Et view er en visuell representering av sin modell. Den fungerer som et "presentasjonsfilter" ved å fremheve visse attributter, og undertrykke andre.
Kontroller - En kontroller er koblingen mellom bruker og systemet. Den passer på at bruker får relevante views på relevante deler av skjermen. Kontrolleren passer også på at brukeren har menyer eller andre måter å gi kommandoer og data på. Etter å ha mottatt input fra bruker vil kontrolleren så sende dette til de relevante viewene.

I kontekst av web development som et eksempel, kan **Model** sies å være HTML biten, **View** vil være CSS biten og **Kontroller** vil være nettleseren som bruker ser. [25] [26] [27]

2.4 Spring konsepter og arkitekturer

Dette delkapittelet tar opp konsepter som er relevante når man arbeider med rammeverket Spring.

2.4.1 Dependency Injection

Dependency Injection, DI, en form av Inversion of Control-prinsippet, IoC, er en teknikk hvor man komponerer objekter med avhengighetene de skal holde. Objektet som mottar avhengigheten blir kalt klienten, mens avhengigheten blir kalt en tjeneste. I ett OOP-språk tar dette gjerne form i at en klasse definerer en konstruktør hvor avhengighetene kan bli gitt, i stedet for at klassen selv definerer hva avhengigheten skal være. En utenforliggende klasse vil stå for å gi tjenesten til klienten. Spring blir på folkemunne blir kalt et DI-rammeverk. [28]

2.4.2 Inversion of Control

Inversion of Control, IoC, er en programmeringsstil, hvor, som ved DI, objekter blir injisert med avhengigheter. Forskjellen fra DI er man at lar annen kode, gjerne et rammeverk, injisere avhengigheten ved kjøretid, i motsetning til at man selv programmerer avhengigheten. [29] [30] [31]

2.4.3 IoC Container

Kjernen til Spring rammeverket er deres IoC container. IoC containeren har ansvaret for å instansiere, konfigurere og komponere sammen avhengighetene. Dermed er det IoC containeren som har kontrollen over hvilke avhengigheter objekter skal ha, ikke objektene selv. [30] [32]

2.4.4 Bean

En Bean er et objekt som blir styrt av Spring IoC Containeren. Et slikt objekt kan annotert med ulike konfigurasjoner som forteller Spring hvordan de skal behandle objektet. En Bean kan være alt fra domene og entitetsklasser til tjeneste og kontrollerklasser.
I Spring bruker man annoteringen "Autowire" til å definere at en Bean skal injiseres med en avhengighet. [33]

2.5 Vue konsepter

Dette delkapittelet tar for seg konsepter som man kommer borti når man utvikler løsninger i Vue.js.

2.5.1 Reaktivitet og databinding

Reaktivitet er ikke et konsept unikt for Vue.js, men er i kjernen for hvordan det, og andre komponentbaserte JavaScript-rammeverker oppfører seg. Reaktivitet er et programmeringsparadigme, hvor man er opptatt av datastrømmer og propagering av endringer i datastrømmene. For et rammeverk som Vue.js, som har ansvaret for presentasjonen av applikasjonen, betyr det hvis dataen det har ansvaret for å vise endrer seg, så skal denne endringen gjengis visuelt.

I Vue.js gjøres dette i praksis ved at man forteller Vue.js at dataen det holder på skal være bundet til en presentasjon av dataen. Endres dataen Vue.js holder på vil presentasjonen reflektere denne endringen. Det defineres en enveis-binding, hvor om dataen endrer seg, så endrer presentasjonen seg, og en toveis-binding, hvor man i tillegg lar endringer i presentasjonen føre til endringer i dataen. Toveis-binding er særlig relevant når man vil ha input fra bruker, slik som i Form HTML-elementer. [34]

2.5.2 Komponenter

Vue.js er et komponentbasert JavaScript rammeverk. Man beskriver mindre, isolerte og gjenbrukbare komponenter, som man kan komponerer sammen til en større løsning. Man vil samle all nødvendig logikk for en oppgave i en komponent. Det definerer en rotkomponent, hvor alle direkte barnekomponenter må registreres. Disse komponentene igjen kan ha barnekomponenter, men bare en foreldrekomponent, altså som en trestruktur.

Disse komponentene definerer man i JavaScript-filer, og lar man i tillegg til å skrive logikk, også definere HTML-elementer som skal injiseres på en side. Dette gjøres i hva Vue.js kaller Template Syntax, en HTML-basert syntax. Det er også mulig å skrive denne syntaxen direkte i en HTML-fil.
I praksis er en Vue.js Komponent en instans av et JavaScript-objekt. [35] [36]

2.5.3 Lasting av komponenter

For å vise en komponent på en side må man gi et HTML-element i en id i en HTML-fil, hvor denne rotkomponent av Vue.js skal lastes inn. I rotkomponenten oppgir man denne IDen til elementet. Har man flere HTML-filer, og man vil bruke Vue.js, må man laste inn en rotkomponent på siden. Dette gjøres gjennom Script-elementet i HTML. [37]

2.5.4 Template Syntax

Vue.js bruker en egen HTML-basert syntax for å beskrive HTML-elementer, samt binde dataen en komponent holder på, til disse HTML-elementene. Man kan legge til spesielle attributter til HTML-elementene, kalt direktiver. Verdien til disse attributtene skal være JavaScript uttrykk. Direktivene brukes til å kunne manipulere nettleserens DOM. For eksempel, man kan ha HTML-elementet med direktivet v-if med uttrykket "present" <p v-if=" present">tekst</p>. I Vue.js instansen finnes det et data-objekt med en Boolean egenskap. Om den er sann, så vises elementet, om den er usann, så vises de ikke.

2.5.5 Single File Components

Vanlige komponenter, som man skriver i JavaScript-filer, har ikke mulighet til å behandle CSS. Derav må CSS defineres i egne filer og legges til ved en Style-elementet i HTML-filen. Samtidig, om man vil gjenbruke komponenter, så kan man ikke importere samme JavaScript-fil som inneholder komponenten flere ganger. Man må i så tilfelle duplisere komponenten og gi den et annet navn. For større prosjekter blir utvikling vanskelig.

Derfor lar Vue.js man definere det man kaller Single File Components, SFC, hvor man bruker en egen Vue-filtype. Disse komponentene lar man samle Template-syntaxen, Vue.js-kode og CSS i en og samme fil. Siden komponentene nå eksistere i filer som Vue.js styrer, kommer man seg rundt problemet med gjenbruket av komponentene og Script-elementet. Vue.js kan nå gi hver komponent et unikt navn, som Vue.js har kontroll på.
Gjennom SFC, så oppnå man SoC, men med at man samler struktur, utseende og oppførsel i samme fil, i stedet for spredt over flere filer. Global CSS kan man fortsatt definere globalt. [38]

2.5.6 Komponentlivssyklus

I Vue.js går en komponent gjennom en livssyklus. Den har sekvensiell rekkefølge av tilstander den kan gå gjennom fra den blir opprettet til den blir destruert. Dette er nyttig om man f.eks. har kode man vil kjøre før en komponent vises eller om man stoppe kode som kjører før komponenten destrueres. [39]

2.5.7 Client Side Routing

Et konsept sterkt knyttet til Single Page Applications er Client Side Routing. I en SPA finnes det bare en HTML-fil, og når man går til en URI innenfor applikasjonen, f.eks. "mittdomene.no/mine-sider", får man ikke returnert minesider.html, men index.html. Med andre ord, så må rutingen til riktig innhold skje i presentasjonslaget. Så når brukeren går til den nevnte URI 'en, så må man i presentasjonslaget definere hva brukeren skal vises, om det så er innhold som faktisk finnes eller om det er en beskjed om at innholdet ikke finnes.

2.6 Sikkerhet

I dette underkapittelet diskuteres det teknologier og verktøy for sikkerhet brukt i oppgaven.

2.6.1 JWT

JSON Web Token, *JWT*, er en åpen standard for å lage JSON-baserte tilgangs-token. Den definerer en kompakt og selvstendig måte for å trygt sende informasjon som et JSON objekt. Informasjonen som sendes kan verifiseres og er trygg fordi den er digitalt signert.

En JSON Web Token består av en header, payload og signatur. Headeren forteller hvilken type token det er og hvilken algoritme som brukes for å hashe. Payloaden inneholder kravene i tokenet. Det er tre typer krav: Registrerte, offentlige og private. Disse kravene er forskjellige verdier som klient eller server kan bruke til å autorisere brukere. Signaturen blir brukt til å verifisere at en beskjed ikke har endret seg på veien, og om en token har en privat nøkkel, blir signaturen brukt for å verifisere at senderen av tokenen er den rette.

JWT lar brukeren slippe å autentisere seg med brukernavn og passord for hver operasjon som krever det. [40]

2.6.2 HTTPS

HTTPS, Hypertext Transfer Protocol Secure, er en utvidelse av HTTP, Hypertext Transfer Protocol. I HTTPS er kommunikasjonsprotokollen kryptert med bruk av TLS, Transport Layer Security.

I tjenester hvor en brukers autentiseringsdata, samt annen data, skal bevege seg over internett, må disse dataene krypteres. Dette for hensyn til brukerens private informasjon og integriteten av den utvekslede dataen. [41]

2.7 Databaser

I dette underkapittelet tar man opp punkter rundt databaser som er relevant for denne oppgaven.

2.7.1 Relasjonsdatabase

En relasjonsdatabase er en form for database hvor relasjonsmodellen blir brukt på data. Dette vil da si å legge forskjellig data, attributter, inn som rader, gruppert i en tabell. En

tabell kan også kalles en relasjon. En rad i en tabell kan identifiseres med en unik nøkkel, kalt en Primærnøkkel. Informasjonen i denne raden skal ha en logisk knytning til denne primærnøkkelen og kombinasjonen av attributtene skal være unik for en primærnøkkel. Dette så man unngår redundans i informasjonen en tabell inneholder. Man kan beskrive knytninger mellom tabeller som forhold. Det finnes tre forhold. De er et-til-et forhold, et-til-mange forhold og mange-til-mange forhold. [42] [43]

2.7.2 Primærnøkkel

En primærnøkkel, også kalt Primary Key, er en identifikator som unikt identifisere en rad i en tabell. Informasjonen i denne raden skal ha en logisk knytning til denne primærnøkkelen og kombinasjonen av attributtene skal være unik for en primærnøkkel. [44]

2.7.3 Sekundærnøkkel

En sekundærnøkkel, også kalt Foreign Key, er en identifikator som unikt identifisere en rad i en annen tabell. Den gir uttrykk for en avhengighet til tabellen den identifiserer. [45]

2.7.4 Et-til-et forhold

Et et-til-et-forhold mellom to tabeller sier at en rad i den ene tabellen skal bare kunne knyttes til én rad i den andre tabellen. Et eksempel kan være at en munn kan bare høre til et menneske, og et menneske kan bare ha en munn. [46]

2.7.5 Et-til-mange forhold

Et et-til-mange forhold mellom to tabeller sier at en rad i den ene tabellen kan knyttes til mange rader i den andre tabellen. For eksempel kan en bokhylle ha mange bøker, men en bok kan bare høre til en bokhylle. [47]

2.7.6 Mange-til-mange forhold

Et mange-til-mange forhold mellom to tabeller sier at flere rader i ene tabellen kan knyttes til flere rader i andre tabellen. For eksempel kan mange skuespillere være i mange filmer, og mange filmer kan ha mange skuespillere. [48]

2.7.7 SQL

SQL - Structured Query Language er språket som blir benyttet for å gjennomføre diverse datamodellering forespørsler i en relasjonsdatabase. SQL er ekstremt kraftig og nyttig for å behandle data i både store og små mengder. [49]

2.7.8 ORM

Objekt-relasjonell mapping, ORM, er en teknikk som muliggjør konvertering av data mellom to forskjellige typesystemer, slik som objekter i objektorientert programmering og relasjoner, eller tabeller i relasjonsdatabaser. I praksis lar dette knytte felter i et objekt til attributter, eller kolonner, i en tabell, hvor et objekts tilstand gjenspeiles som en rad i tabellen. [50]

2.8 Sjakknotasjoner og -filformater

Dette delkapittelet vil fortelle om notasjoner og filformater som brukes til å presentere og strukturere informasjon om sjakkturneringer og sjakkspill.

2.8.1 Portable Game Notation

Portable Game Notation, PGN, er et filformat hvor man kan representere et sjakkspill som ren tekst. Filformatet holder på informasjon rundt selve sjakkspillet, slik som hvilke spillere brukte hvilke brikker, og informasjon hvilke trekk ble gjort hvilken runde og resultatet av spillet. Ble utviklet av Steven J. Edwards. [51]
Et eksempel er vist nedenfor i figur 2.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 {This opening is called the Ruy Lopez.}
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

Figur 2 - Eksempel av en PGN - Viser et ferdigspilt parti

2.8.2 Forsyth-Edwards Notation

Forsyth-Edwards Notation, FEN, er en standardisert notasjon for å beskrive tilstanden av et sjakkspill, på en linje med ASCII-karakterer. En slik tekstlinje kalles en FEN-streng. Ble først utviklet av David Forsyth på nittenhundretallet, før det ble videre utviklet av Steven J. Edwards for å gjøre notasjonen lesbar for datamaskiner. [52]
Et eksempel er vist nedenfor i figur 3.

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Figur 3 - Eksempel av en FEN-streng - Viser startposisjon på sjakkbrett

2.8.3 TRX

TRX er et filformat brukt av programvaren Turneringsservice. Filformatet følger en XML-struktur. TRX inneholder all informasjon fra en turnering som brukerne av Turneringsservice har fylt inn i programvaren. Filen er først og fremst ment til å bli lastet opp til Turneringsservice sin nettjeneste, hvor informasjonen for turnering blir gjengitt.

2.9 Språk, Rammeverk og teknologier

Dette delkapittelet beskriver språk, rammeverker og teknologier relevante for dette prosjektet.

2.9.1 Java

Java er et objektorientert språk som ble utviklet av James Gosling som arbeidet for Sun Microsystems, som senere ble kjøpt opp og er nå eid og styrt av Oracle. Prinsippet Java er bygd opp på er at den skal kunne bli skrevet en gang, og kjørt hvor som helst, "*write once, run anywhere (WORA)*". Java er laget slik at etter den er kompilert vil den så lenge det finnes en JVM (Java Virtual Machine) kunne kjøres på systemet, uavhengig av hvilket system det er.

2.9.2 JavaScript

JavaScript, også forkortet til JS er et høynivå skriptspråk, som kjørers i JavaScript-kjøretider. Hver moderne nettleser har en innebygd JavaScript-kjøretid. Noen verktøy, slik som Node.Js bruker en slik JavaScript-kjøretid til å kunne kjøre JavaScript på servere. Dette muliggjør JavaScript på backend. Språket er de facto-språket for å gi nettsider interaktivitet.

2.9.3 HTML

HTML står for HyperText Markup Language. Det er et språk for strukturering og beskrivelse av innhold på en nettside. Elementer blir brukt for å markere innhold. Elementer kan være semantiske eller ikke. Semantiske elementer skal beskrive at innholdet har en mening eller kontekst, samt gi det struktur. Slik som paragraf-elementet <p> skal beskrive at innholdet er en paragraf i en tekst. Ikke-semantiske elementer beskriver ikke noe innhold, men er bare brukt for inndeling av innhold. Det er ikke et programmeringsspråk da det ikke har støtte for noe logikk.

2.9.4 CSS

CSS står for Cascading Style Sheets, og er et språk for å beskrive presentasjonen av innholdet i språk slik som HTML.

2.9.5 Rammeverker og teknologier for Java

I dette delkapittelet beskriver man rammeverker og teknologier som relevante innenfor Java for denne oppgaven.

2.9.5.1 Spring Rammeverket

Spring er et åpen-kilde rammeverk laget for å kunne bli benyttet og brukt i Java applikasjoner. Det er som nevnt i 2.4.1 og 2.4.2 et Dependency Injection-rammeverk, som bruker Inversion of Control-prinsippet, hvor rammeverket tar ansvaret for å injisere avhengigheter i kjøretid. Spring rammeverket er svært sterkt modulisert, hvor man kan legge til flere moduler omtrent på et Plug-and-Play vis. [53]

2.9.5.2 Spring Boot

Spring Boot er en adapsjon av Spring rammeverket, men for å tillate utviklere til å enkelt og raskt kunne få kjørt opp en applikasjonsserver. Spring Boot er designet for å minske unødvendig pakker og bibliotek, også kjent som dependencies. Dette er for å minske unødvendig belastning og tillate utviklere i å enklere kunne velge og vrake og kun benytte de man trenger. Spring Boot kommer med en egen applikasjonsserver, samt tar seg av konfigureringen av den og moduler som følger med, men man må da forholde seg til valgene som er tatt for Spring Boot. Spring Boot definere også sine egne pakker, kalt Starters, som inneholder mye brukte avhengigheter. Disse pakkene er ment å gi en utviklere en basis av avhengigheter for å løse visse utfordringer. [54] [55] [56]

2.9.5.3 Spring MVC

Spring MVC er et Spring rammeverk bygd rundt Model-View-Controller mønsteret. Det er dette rammeverket som har støtte for REST-arkitekturstilen. [57]

2.9.5.4 Spring Security

Er et autentiserings- og autoriseringsrammeverk, som kan innlemmes som en egen modul i en Spring applikasjon. Har beskyttelse mot mange typer angrepsvektorer, slik som Cross Site Request Forgery, hvor de aller fleste er aktivert som standard. [58] [59]

2.9.5.5 Spring Data JPA

Spring Data JPA er et rammeverk som pakker inn en JPA-implementasjon. Den forenkler mye av seremonien rundt interaksjonen med JPA, slik at man kjappere og enklere kan kunne håndtere data. Spring Data JPA tillater utviklere til å kun måtte skrive sine «repositories» som Interfaces med ønskede metoder, men så vil Spring automatisk implementere disse. Dette gjør at man for de støttede metodene slipper å skrive hele database kallet og lignende.

2.9.5.6 Spring Initializr

Spring Initializr er et verktøy Spring tilbyr hvor en ekstremt enkelt og fort kan få satt opp prosjektet sitt. På nettsiden kan man velge diverse hvilke byggesystem en vil benytte, Maven eller Gradle. Hvilket språk, Java, Kotlin eller Groovy og hvilken versjon av Spring Boot man vil benytte og mer. Her kan man også enkelt gå gjennom og legge inn hvilke av de mange avhengighetene man trenger og ønsker å ha i deres applikasjoner. [60] [61] [62]

2.9.5.7 JPA

JPA - Jakarta Persistence API (tidligere Java) er en spesifikasjon laget for å håndtere styringen av data i Java applikasjoner. JPA tilbyr et abstraksjonsnivå mellom Java objekter, og relasjonsdatabaser. Istedenfor vil verktøy som Hibernate nevnt i 2.9.5, implementere JPA sine spesifikasjoner. [63]

2.9.5.8 Hibernate

Hibernate er et ORM (Objekt-relasjon modell) verktøy innenfor Java som blir benyttet for å gjennomføre handlinger via mapping mellom objektorienterte modeller og en relasjonsdatabase. Hibernate implementer JPA spesifikasjonen. [64]

2.9.5.9 Maven

Maven er et byggesystem som blir brukt for å sette opp prosjekter for Java. Maven er basert på en prosjekt-objekt-modell (POM) og kan legge til .jar filer og andre avhengigheter ved oppstarten av et prosjekt. Med Maven får man også med informasjon og dokumentasjon om prosjektet. [65] [66]

2.9.6 Rammeverker og biblioteker for JavaScript og CSS

I dette delkapittelet nevner vi biblioteker og rammeverker for JavaScript CSS som er relevant for denne oppgaven.

2.9.6.1 Vue.Js

Vue.js er et fleksibelt komponentbasert JavaScript-rammeverk som bare fokuserer på presentasjonslaget i en arkitektur. Det har støtte til å brukes i alle størrelser prosjekter, fra de minste hvor man trenger minimalt med funksjonalitet det tilbyr, til de største Single Page Application-prosjektene. [67]

2.9.6.2 Chess.js

Chess.js er et JavaScript-rammeverk for analysering av sjakktrekk og sjakkbrett. De støtter blant annet validering av sjakktrekk, validering av PGN og generering av FEN-strenger fra PGN. [68]

2.9.6.3 Chessboard.js

Chessboard.js er et JavaScript-bibliotek for vise sjakkbrett, med animasjoner for flytting av brikker. Har blant annet funksjonalitet for å bevege brikker basert på FEN-strenger gir som input. [69] [70]

2.9.6.4 Axios

Axios er et JavaScript-bibliotek, som lar man kjøre en HTTP-klient, enten ved bruk av HTTP forespørsler om man bruker Node.js, eller ved bruk av XMLHttp-forespørsler om man bruker Axios i nettleseren. XMLHttp-forespørsler har den fordelen at man unngår en oppdatering av nettsiden når man får tilbake responsen. Biblioteket støtter også asynkrone HTTP kall, automatisk transformasjon fra JSON til JavaScript objekter, og fange og utføre endringer på forespørsler og responser. Det siste punktet forenkler den nødvendige logikken for autentisering og autorisering fra JavaScript. [71] [72] [73]

2.9.6.5 Bootstrap

Bootstrap er et CSS-rammeverk, som tilbyr ferdigstilte komponenter eller maler for komponenter, for det meste av grafiske komponenter på nettsider, slik som knapper, forms m.m. Komponentene og malene er designet for å kunne brukes på desktop-nettleser og mobil-nettlesere. [74] [75]

2.9.7 Verktøy, rammeverker og biblioteker for Vue.Js

I dette delkapittelet beskriver vi verktøy, rammeverker og biblioteker som er nyttige for utvikling med Vue.js

2.9.7.1 Vue CLI

Vue CLI er et kommandolinjeverktøy for å opprette og styre Vue.js-prosjekter. Det lar man velge avhengigheter man vil ha, samt inkluderer JavaScript-kompilatoren Babel, modul-bundleren Webpack, kodeformatøren Prettier [76] og kodeanalyseverktøyet ESLint. [77] Vue CLI gjør det enkelt å starte et Vue.js-prosjekt. Særlig verktøyene Babel [78] og Webpack [79]er viktige. Babel lar man skrive JavaScript ES6 syntax, og så tar Babel av seg å gjøre denne kompatibel med tidligere syntax for eldre nettlesere. Webpack tar alle dine applikasjonsfiler og pakker de sammen til en eller få filer, med den fordel at man får redusert antall HTTP-forespørsler som er nødvendig for å hente all koden for applikasjonen din. Med Vue CLI kan man opprette de mindre enkle prosjekter og de større Single Page Application-prosjektene. Vue CLI er laget av samme organisasjon som laget Vue.js. [80]

2.9.7.2 Vue Router

For å kunne navigere mellom forskjellige deler i en Single Page Application, trenger man Client Side Routing. I en Vue.js Single Page Application kan man ti bruke Vue Router. Man kan definere hvilke komponenter samsvarer med hvilke URL-er, så tar Vue Router seg av navigeringen. Vue Router er laget av samme organisasjon som laget Vue.js. [81] [82] [83]

2.9.7.3 Vuex

Vuex [84] er et bibliotek og tilstandsbehandlingsmønster, også kalt state management, for Vue.js Vuex fungerer på måten at det blir opprettet et sentralisert lager, «store» er betegnelsen som blir brukt i koden. I dette lageret til man kunne gjøre diverse håndteringer relatert til informasjon som skal måtte bli lagret og benyttet på tvers av applikasjonen. Vuex beskriver tre typer begreper man kan bruke for å håndtere tilstand. Disse er typene getters, mutations og actions. Metodene av typen getter lar hente en tilstand. Mutations lar deg mutere tilstanden. Actions metoder lar deg utføre mutations og støtter asynkrone kall, noe som gjør det fint for å gjennomføre f.eks. ved login funksjonalitet. Vuex er laget av samme organisasjon som laget Vue.js.

2.9.7.4 Vuelidate

Vuelidate [85] er et tredjeparts bibliotek for validering av data. Vuelidate inneholder mange forhåndsbaserte regler, alt fra nødvendig, epost til maks og minimum lengde, og mange flere. Disse kan bli importert og mappet til en modell i Vue.js sin datamodell for å sørge for at disse tekstfeltene og lignende sin data blir fylt ut som ønsket. I motsetning til Vuelidate, så validerer dataene Vue.js holder på i en komponent, så validerer andre valideringsrammeverker validerer innholdet i Vue.js sin Template Syntax. [86]

3 MATERIALER OG METODE

3.1 Organisering av prosjekt

Dette delkapittelet beskriver hvordan prosjektet og gruppen ble organisert.

3.1.1 Prosjekt Team

En var sekretær, ansvarlig for kommunikasjon med veileder og oppdragsgiver og organisering av møter med dem. Gruppen var sidestilte i avgjørelser. Gruppen var på tre medlemmer, så avgjørelser ble tatt av flertall. Gruppen hadde ingen valgt prosjektleder.

3.1.2 Møter med veileder

Gruppen arrangerte møter med veileder annenhver torsdag I forkant av hvert av disse møtene sende gruppen en framdriftsrapport til veilederen som innhold oppdatering om hva gruppen hadde arbeidet med og noen nøkkelpunkter for tankene rundt den da ferdige arbeidsperioden. I møtet hadde vi en dialog med veileder om eventuelle tanker og spørsmål relatert til prosjektet. Møtene ble holdt over video- og lydplattformen Zoom, og det ble tatt møtenotater under møtene. Møtereferater ligger som vedlegg H.

3.1.3 Møter med oppdragsgiver

Gruppen kalte inn til møter oppdragsgiver annenhver torsdag. I disse møtene gikk vi gjennom hva vi hadde arbeidet med og oppnådd i løpet av sprinten. Her hadde vi en dialog med oppdragsgiver om hvordan vi hadde gjort de forskjellige oppgavene og hvordan vi tenkte å gå frem. Ettersom vår oppdragsgiver har teknisk bakgrunn bisto han også som hjelp på den tekniske delen i tillegg. Vi hadde også samtaler med dem om hva de ønsket vi skulle prioritere i neste arbeidsperioden, men oftest hadde oppdragsgiver ingen spesifikke tanker og lot gruppen stå mer fritt til å prioritere selv. Møtene ble holdt over video- og lydplattformen Zoom, og det ble tatt møtenotater under møtene. Møtereferater ligger som vedlegg H.

3.1.4 Møter innad i gruppen

På grunn av pandemien har gruppen i all hovedsak arbeidet hjemmefra og har i den forbindelse oftest sittet i lydsamtaler sammen i arbeidstiden for å enkelt kunne kommunisere med hverandre. Som følge av dette har ikke gruppen hatt mange spesifikke møter det har blitt innkalt til annet enn møter i forkant av de tidligere nevnte møtene med veileder og oppdragsgiver, samt planlegging av sprint. Møtene ble holdt over video- og lydplattformen Discord.

3.2 Smidig utviklingsmetodikk

I dette delkapittelet går man over hvordan gruppen fulgte og gjennomførte en Smidig utviklingsmetodikk.

Gruppen jobbet etter en Smidig utviklingsmetodikk, nærmer sagt Scrum. Gruppen arbeidet i to ukers Sprinter, hvor disse gikk fra torsdag til torsdag første halvdel. Etter de første ukene ble perioden endret til start fredag og slutt fredag. Dette for å få bedre tid til de ulike Sprint-møtene, som fikk for lite tid for på torsdagene.

3.2.1 Sprint planleggingsmøte

I starten av hver sprint gjennomførte gruppen et Sprint-planleggingsmøte. I dette møtet gikk gruppen gjennom arbeidsoppgavene som var øverst på Produkt-backloggen. Gruppen diskuterte hva vi trodde skulle til for å utføre oppgaven, for så estimere de. Så ble arbeidsoppgaven delt opp i mindre Sprint-arbeidsoppgaver. Om oppdragsgiver hadde nye ønsker eller prioriteringer, var de lagt til og omprioritert i dette møtet. Oppdragsgiver kom sjeldent med ønsker eller prioriteringer.

3.2.2 Sprint gjennomgangsmøte

I slutten av hver sprint hadde vi på torsdagene møte med oppdragsgiver for å oppdatere ham med hvilke fremskritt gruppen hadde gjort, og om der var noen ønsker for hva man skulle jobbe med i neste sprint. Basert på tilbakemeldingene gikk gruppen som nevnt videre og planlagte kommende sprint i ett Sprint planleggingsmøte.

Sprint gjennomgangsmøterapporter er lagt til som vedlegg G.

3.2.3 Sprint Retrospektivmøte

Etter endt Sprint, avholdt gruppen Sprint Retrospektiv møter. Disse ble brukt til å ta opp hvordan gruppen mente de presterte og hva som kunne forbedres. Sprint retrospektivmøtenotatene er lagt til som vedlegg F.

3.2.4 Daglig Scrum-møte

Dette var en del av Scrum gruppen ikke fulgte. Gjennom pandemien har gruppen kommunisert over Discord gjennom de avtalte arbeidstidene. Her delte gruppen uansett hva de jobbet med og, om de stod fast, spurte om hjelp. Derav var nytten av å sette av en 5-15 minutters periode for et daglig Scrum-møte liten.

3.3 Verktøy for prosjektstyring

I dette delkapittelet beskriver man hvilke verktøy ble brukt prosjektstyring.

3.3.1 Jira

Jira er et produktstyringsverktøy som følger den smidige utviklingsmetodikken for programvareutvikling, mer spesifikt, Scrum. Produsert av Atlassian, så brukes Jira hovedsakelig til sporing av problemer og prosjektledelse. Man kan blant annet planlegge og starte/slutte sprinter, følge med på medarbeideres framgang, se på utgaver osv. Jira ble brukt i vår utøvelse av Scrum. [87]

3.3.2 Confluence

Confluence er et samarbeidsverktøy laget av Atlassian. Verktøyet inneholder maler for mange forskjellige typer dokumenter. Confluence har en tett knytting til Jira og andre Atlassian produkter. I en programvareutviklingkontekst kan man blant annet bruke Confluence til å opprette og samle kravspesifikasjoner, gjerne med brukerhistorier, og knytte disse til arbeidsoppgaver i Jira. Man kan også samle andre dokumenter slik som møtenotater, dele notater, dele kunnskap og planlegge prosjekter. Gruppen brukte Confluence, sammen med Jira. Confluence ble, sammen med Discord, hvor vi lagret prosjektrelaterte dokumenter. [88]

3.3.3 Discord

På grunn av Korona-pandemien ble arbeidet utført omtrent bare på digitale plattformer. Discord ble vår daglige samarbeidsplattform. Discord er et verktøy hvor man kan både skrive og snakke til hverandre, enten direkte eller innenfor brukerstyrte servere. På en server kan man opprette tekstkanaler eller video- og lydkanaler. Denne funksjonaliteten var veldig attraktivt i gruppens tilfelle. Dette siden det gjør det mulig å organisere kanaler, og dermed å enkelt navigere til forskjellige dokumenter, lenker til dokumenter og lenker til andre ressurser som ble brukt gjennom semesteret. Når gruppen skulle kommunisere verbalt var det så enkelt som å trykke på en snakke-kanal som konstant holder seg oppe. Verktøyet det kan nærmest sammenlignes med er Slack.

3.3.4 Git Flow

Under selve utviklingen fulgte gruppen Git Flow for organisering av grener og integrasjon av kode. Gruppen tok ingen form for sjekk av kode før integrasjon av kode.

3.4 Planlegging

Dette delkapittelet tar for seg hvilken planlegging ble utført og i hvilken form.

3.4.1 Forprosjektsrapport

Før gruppen satt i gang med bacheloroppgaven, ble det utført en obligatorisk forprosjektperiode og rapport. I denne ble blant annet problemstilling og mål for oppgaven definert. På bakgrunn av diskusjoner med arbeidsgiver og oppdragsgiver i denne perioden, ble det planlagt en periode for informasjonsinnsamling. Dette er diskutert i punktet "Informasjonsinnsamling - utført og planlagt" i forprosjektrapporten. Forprosjektsrapport og presentasjon av forprosjekt er lagt ved som vedlegg I.

3.4.2 Informasjonsinnsamling

Tidlig i oppgaveperioden utførte gruppen to perioder med informasjonsinnsamling. I den ene perioden ble det utført en analyse av flere nettjenester for spilling på og vising av digitale sjakkbrett. Hensikten var å bli nærmere kjent med hvordan sjakkpartier og sjakkturneringer ble presentert på nett. Funn i denne analysen ble brukt videre i design av nettsiden. I den andre perioden ble det foretatt en analyse av programmeringsspråk, rammeverker og applikasjonsserver gruppen så for seg som relevante. Funn i denne analysen spilte inn på valg av teknologier. Analysene er lagt til som vedlegg E.

3.4.3 Wireframes

Wireframes er en måte å designe en nettside på fra bunnen av. Det blir som regel brukt for å få en idé av hvordan layoutet skal være og hva som trengs av innhold på nettsiden. Wireframes er typisk noe av det første som blir gjort i prosessen av å lage en nettside. Det mangler som regel grafiske elementer som farge og stil, og fokuserer heller på funksjonalitet.

Gruppen brukte wireframes for å gi et første utkast av presentasjonslaget til løsningen. Disse ble presentert til oppdragsgiver, hvor han gav positive tilbakemeldinger. Wireframes er lagt til som vedlegg C. [89] [90]

3.4.4 Kravspesifikasjoner

Kravspesifikasjoner for løsningen ble definert tidlig i prosjektet. Disse ble skrive som eposer, og som brukerhistorier, fra synspunkter til flere roller vi definerte. Rollene var en seer, turneringsorganisator og deltaker. Det ble ordnet en overordnet kravspesifikasjon. Her ble større deler eller komponenter løsningen skulle ha, skrive som eposer. Så ble det skrive egne kravspesifikasjoner for hver av disse større delene, hvor kravene ble beskrive som brukerhistorier. Ett eksempel er at gruppen beskriver eposet "Vising av live sjakk" hvor brukerhistorien beskriver "Som en seer så vil jeg se minst et sjakkbrett med automatiske oppdateringer så jeg får med meg hvordan partiet skrider frem." En annen brukerhistorie kan så være "Som en seer, vil jeg bli vist hvem som er vinneren, så jeg vet hvilken spiller vant partiet". Kravspesifikasjonsdokumenter er lagt til som vedlegg D.

3.5 Teknologier for utvikling

Dette underkapittelet beskriver teknologier som ble brukt i løsningen av denne oppgaven, samt gir grunn for valgene av teknologiene.

3.5.1 Presentasjonslaget - Frontend

Denne delen beskrive hvilke språk, rammeverker, biblioteker og verktøy som ble brukt for utvikling av presentasjonslaget.

3.5.1.1 JavaScript

For utvikling av presentasjonslaget ble JavaScript valgt som språk. Løsningen innebærer en nettside i form av en Single Page Application med Client Side Rendering, og da er det ikke mange valgene man har. Andre valg kunne vært TypeScript, som er et spesialisert språk bygget på toppen av JavaScript, som blant annet innfører statisk typing av data. Gruppen hadde i forkant av oppgaven ikke så stor kjennskap til JavaScript. Derav ble det vurdert som uklokt å starte rett på TypeScript, uten å bli kjent med JavaScript. Gruppen prøvde å følge ES6, ECMAScript 6, kompatibel kode.

3.5.1.2 Vue.js

Å utvikle en Single Page Application-lik funksjonalitet i ren JavaScript er en bacheloroppgave i seg selv. Så for å utvikle en SPA, falt valget på Vue.js, nærmere sagt Vue 3, den nyeste versjonen av rammeverket. Valget av Vue.js ble tatt for to grunner. Den ene grunnen var at oppdragsgiver allerede hadde utviklet en liknende tjeneste selv, som gruppen fikk tilgang til. Dette lot gruppen både hente inspirasjon fra løsningen, samt gjenbruke noen av komponentene som allerede var utviklet. Den andre grunnen var at gjennom analysen, beskrevet i 3.4.2, viste at Vue.js var blant de raskeste og mest brukervennlige presentasjonrammeverkene i JavaScript-universet. Valget av å bruke versjon 3 av Vue.js gav oss noen utfordringer som blir beskrevet senere.

3.5.1.3 Vue CLI

Vue CLI er gruppen brukte for å opprette og styre prosjektet. Vue CLI forenkler opprettelsen av Single Page Application-prosjekter. Verktøyet er standard for oppretting av Vue.js-prosjekter.

3.5.1.4 Babel

Babel er støttet av og kommer med når man oppretter et Vue.js-prosjekt med Vue CLI. Babel muliggjør utvikling med ES6 kompatibel kode, også for nettlesere som ikke støtter det fullt ut. Babel ble beskrive i punkt 2.9.10.1.

3.5.1.5 Webpack

Webpack er støttet av og kommer med når man oppretter et Vue.js-prosjekt med Vue CLI. Webpack hjelper med å gjøre innlasting av applikasjonen raskere da man reduserer nødvendig HTTP-kall, gjennom å pakke alle deler av applikasjonen i en eller få JavaScript-filer. Webpack ble beskrive i punkt 2.9.10.1.

3.5.1.6 Vue Router

Siden nettsiden skulle utvikles som en Single Page Application, trenger man å håndtere Client Side Routing. I Vue.js-prosjekter er standardløsningen å bruke Vue Router. Gjennom opprettelsen av prosjekt med Vue CLI får man valg om å legge til Vue Router.

3.5.1.7 Vuex

Vuex ble brukt til å håndtere global tilstand i prosjektet. Et slikt verktøy forenkler mye behandling av brukerinformasjon og login-status på tvers av prosjektet. Gjennom opprettelsen av prosjekt med Vue CLI får man valg om å legge til Vuex.

3.5.1.8 Vuelidate

Det å inkludere et valideringsbibliotek som Vuelidate, gjør validering av data mye mer utviklervennlig. De gir utviklere en enklere API for inputvalidering, samt gjemmer bort mye av koden nødvendig for å implementere det. At valget falt på Vuelidate kom av nødvendighet, da det var et av få kjente valideringsbibliotek med støtte for Vue 3.

3.5.1.9 Bootstrap

CSS-biblioteker som Bootstrap tilbyr mange visuelle komponenter og maler, som er laget med brukervennlighet, universell utforming og estetikk i tankene. Det lar utviklere fokusere mindre på visuelt design og mer på funksjonalitet. Som med Vuelidate, er støtten for Vue 3 blant slike biblioteker ikke utbredt.

3.5.1.10 Chess.js

Chess.js-rammeverket ble brukt for å kunne behandle og validere PGN-informasjon og genere FEN-strenger. Oppdragsgiver fortalte gruppen om biblioteket.

3.5.1.11 Chessboard.js

Chessboard.js-biblioteket ble brukt for å presentere tilstanden på sjakkpartier. Chessboard.js lar støtter animasjon av trekk fra en brett-tilstand til en annen, ved å mate det en FEN-streng. Oppdragsgiver fortalte gruppen om biblioteket.

3.5.1.12 Visual Studio Code

Visual Studio Code, VS Code, er en teksteditor utviklet av Microsoft. Gruppen benyttet denne i arbeidet med presentasjonslaget ettersom VS Code har god støtte for arbeid

med rammeverkene vi valgte å benytte. VS Code har et stort fellesskap som utvikler og legger ut diverse ekstra pakker som gjør for en mer effektiv koding.

3.5.1.13 Node Package Manager

Node Package Manager, NPM, er et pakkesystem for JavaScript. NPM er et system hvor man kan hoste pakker for JavaScript og som fungerer som et repository hvor man kan laste ned og inn nødvendige pakker inn i sitt prosjekt. Dette ble brukt mye i prosjektet da gruppen brukte en del forskjellige pakker for å løse problemer og få diverse funksjonaliteter.

3.5.1.14 Node.js

Node.Js er en JavaScript-kjøretid, hvor kjøretiden lever i en servertjeneste. Node.js gir JavaScript utviklere noen fordeler ved utvikling. Det gir utviklere en lokal server hvor koden deres kan kjøres og servert og derav får teste ut endringer direkte på egen maskin. Samtidig støtter serveren konseptet Hot Reload, hvor når man lagrer en fil mens serveren kjører, så blir prosjektet lastet inn på nytt, uten at man må stoppe og starte serveren. Dette gjør utviklingsprosessen mye raskere.

3.5.2 Datalaget - Backend

Denne delen beskrive hvilke språk, rammeverker, biblioteker og verktøy som ble brukt for utvikling av backend.

3.5.2.1 Java

Java var språket som ble valgt for utvikling på backend. Det er et stabilt og allsidig språk.   Gruppen er godt kjent med språket og kunne dermed bruke denne kunnskapen i prosjektet. I tillegg hadde oppdragsgiver brukt Java og rammeverket Spring for en tidligere løsning som gruppen kunne ta i bruk. Gruppen brukte versjon 11 av Java, da dette er den seneste versjonen med langtids støtte og det er mange ressurser tilgjengelig for denne versjonen.

3.5.2.2 Spring Rammeverket

Spring er industristandarden for Java rammeverker. I og med det er såpass utbredt for Java-utvikling så gruppen på det som fordelaktig å lære om og lære å bruke rammeverket.

3.5.2.3 Spring Boot

Spring Boot ble valgt som grunnlag for applikasjonen, for å raskt komme gang med utvikling. Spring Boot kommer med blant annet en applikasjonsserver hvor tjenesten kan kjøre. Standard applikasjonsserver som er inkludert er Tomcat. Gruppen brukte denne.

3.5.2.4 Spring Web

Spring Web er en Spring Boot starter, en pakke med avhengigheter for grunnleggende funksjonalitet, som inneholder Spring rammeverket MVC og derav REST-arkitekturstøtten for en Spring Boot applikasjon. Denne ble inkludert for nettopp den støtten.

3.5.2.5 Spring Security

Spring Security ble lagt til for dens støtte for autentisering og autorisering, samt innebygde sikkerhetstiltak.

3.5.2.6 Spring Data JPA

Spring Data JPA forenkler interaksjonen med databaselaget, og ble tatt med på grunn av denne forenklingen. Inneholder ORM-en Hibernate.

3.5.2.7 Sprint Initializr

Sprint Initializr ble brukt for å opprette et Spring Boot prosjekt. Vi valgte alle de nevnte Spring rammeverkene nevnt i denne seksjonen. Når man bruker denne tjenesten, er man også sikker på at de ulike Spring rammeverkene man trekker inn er kompatible med hverandre.

3.5.2.8 Hibernate

Hibernate ble brukt som ORM. Følger med som standard ORM i Spring Data JPA.

3.5.2.9 PostgreSQL

Gruppen valgte å benytte PostgreSQL for database. PostgreSQL er veldig utbredt, og som alle på gruppen har brukt ved tidligere anledninger i prosjekt i andre fag. Det har god støtte for flere operativsystemer, noe som var veldig viktig da gruppen brukte flere forskjellige operativsystem innad i utviklingsteamet, samt på hosting-delen. PostgreSQL-databasen ble kjørt i en Docker container.

3.5.2.10 Maven

Maven ble brukt som bygge- og prosjektstyringssystem. Det er en industristandard for Java-prosjekter, samt det er det gruppemedlemmene er kjent med.

3.5.2.11 IntelliJ

Gruppen valgte å benytte IntelliJ IDE-en for utvikling av backend. Gruppen har via. studentlisens tilgang til dens mest funksjonsrike versjon. Denne versjonen har stor støtte for funksjonaliteter som tillot gruppen i å kunne effektivt arbeide med utviklingen. IntelliJ har god innebygd støtte for Maven, Git og Docker og har databaseverktøy inkludert.

3.5.3 Andre verktøy og teknologier

Denne delen andre verktøy og teknologier som ble brukt for utvikling av løsningen

3.5.3.1 Git

Git var verktøyet gruppen brukte for versjonskontroll.

3.5.3.2 Github

Github ble benyttet som sentral repository.

3.5.3.3 Postman

Postman er et sterkt verktøy gruppen benyttet for å teste APIen under utvikling. Den gjør at man enkelt kan gjennomføre alle sjekker man trenger å gjennomføre.

3.5.3.4 Turneringsserivce

Turneringsservice er en norskutviklet programvare for organisering og styring av sjakkturneringer og er mye brukt i norske sjakkturneringer. Dette programmet vil kjøre på PCer hos Aalesund Schaklag. Det lagrer informasjon om turneringene i et eget filformat, TRX. Det var fra denne filen gruppen ville hente informasjon rundt sjakkturneringene. [91]

3.5.3.5 DGT LiveChess

DGT LiveChess er en programvare for å digitalt vise sjakkpartier som foregår på digitale sjakkbrett. Ved turneringer vil en PC med denne programvaren være koblet til de digitale sjakkbrettene og hente informasjon fra dem. Programvaren lagrer hvert parti i en egen PGN-fil. Disse filene blir kontinuerlig oppdatert etter hvert som partiene skrider frem. Det var fra denne filen gruppen ville hente informasjon om sjakkpartier. [92]

3.5.3.6 Statisk kodeanalysering og kodeformatering

Verktøy for statisk kodeanalysering, såkalte lintere, og kodeformateringsverktøy ble brukt for både frontend og backend. For frontend ble ESLint brukt som linter og Prettier for automatisk kodeformatering ved lagring av filer. Disse legges til og konfigureres gjennom Vue CLI ved opprettelse av prosjektet. For backend ble Sonarlint brukt som linter og IntelliJ sin innebygde autoformattering for kodeformatering. Med disse verktøyene blir en standard for hvordan kode skal se ut satt, og gruppen fulgte anbefalingene fra linterne og lot kodeformateringsverktøyene ta seg av formattering, så gruppen slapp å bruke tid på dette.

4 RESULTATER

Dette kapittelet har som hensikt å vise hvilken funksjonalitet gruppen fikk på plass og hvordan teknologier, design og arkitektur ble implementert.

4.1 Funksjonalitet for sluttbruker

I dette underkapittelet vil gruppen vise hvilken funksjonalitet som er lagt til løsningen fra en sluttbrukers perspektiv.
Løsningen vår er som sagt en informasjonsskjerm for sjakkturneringer. Denne løsningen var ment å fungere for alle som bruker programvaren Turneringsservice og DGT LiveChess, herunder Aalesund Schaklag. Vi vil under vise hva som er mulig å gjøre i tjenesten og beskrive hvilke behov det løser.

4.1.1 Visning av turneringer

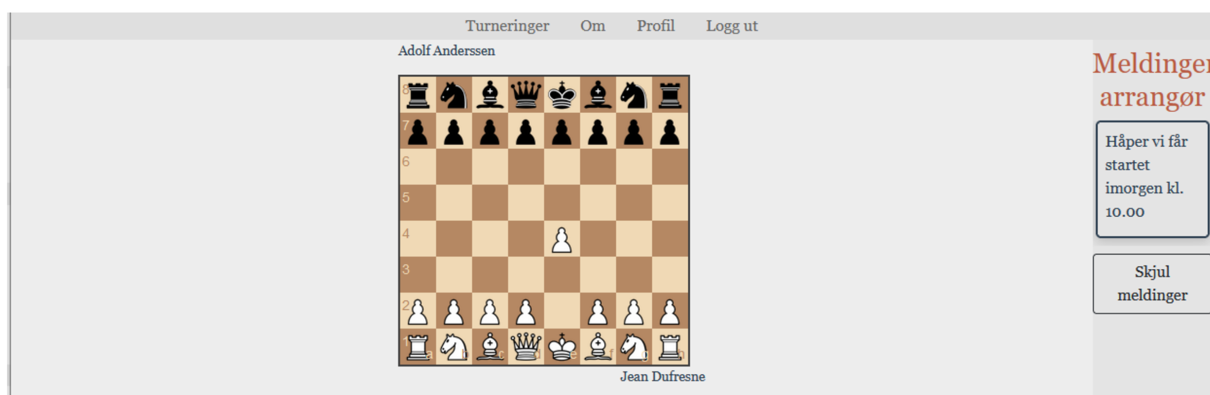
Siden dette er en tjeneste som skal brukes av nevnte programvarer, så trenger man da en side for å vise hvilke turneringer som finnes i systemet, så brukere får navigert til den siden de er etter.

Figur 4 - Visning av turneringer

Vi kom frem til at det var naturlig å dele visningen av disse inn i pågående, kommende og fullførte turneringer, som vist i figur 4. Dette gjør det for eksempel lettere for publikum å finne en som pågår, samt gi turneringsorganisatorer muligheten til å finne frem til turneringer de har planlagt frem i tid. Trykker brukeren på en av turneringene blir brukeren navigert videre til en turneringsskjerm, samt nødvendig informasjon for henting av turneringsinformasjon blir sendt med.

4.1.2 Turneringsskjerm

Det viktigste for denne løsningen er turneringsskjermen, her vist i figur 5. Den har som hensikt å vise å vise tilstanden på digitale sjakkbrett og vise meldinger fra arrangørene av turneringen. Dette gir publikum en oppdatering om en turnering. Turneringsskjermen er en gjenbrukbar Vue.js komponent, hvor informasjon sendt med fra visningen av turneringer brukes til å hente informasjon om den spesifikke turneringen.



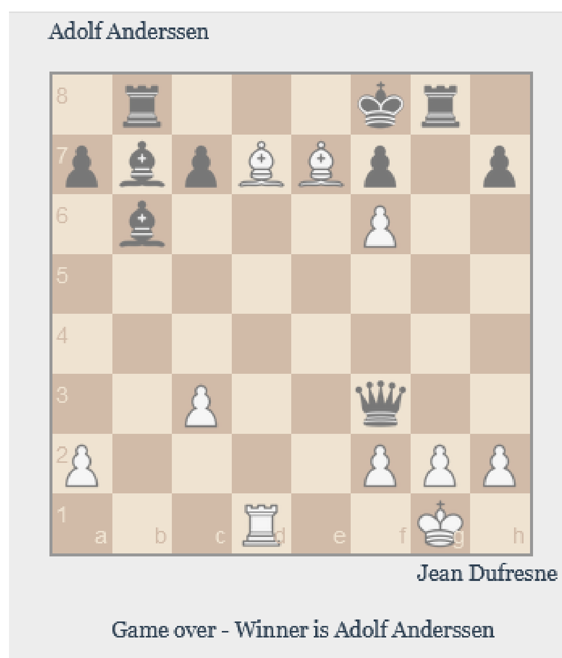
Figur 5 - Visning av turneringsskjerm

4.1.2.1 Visning av sjakkpartier

Visningen av sjakkpartier er implementert ved hjelp av hovedsakelig to JavaScript-biblioteker og rammeverker. Disse er Chess.js og Chessboard.js. En Axios-tjeneste på frontend henter PGN-informasjon fra backend og databasen gjennom et REST-API. Denne PGN-informasjonen blir matet til Chess.js. Chess.js lar man produsere en FEN-streng basert på denne PGN-informasjonen. Denne FEN-strengen blir så gitt til Chessboard.js som tar seg av presentasjonen av sjakkbrettene. Chessboard.js kan så animere et trekk basert på denne FEN-strengen. Denne funksjonaliteten looper på frontenden, slik at eventuelle endringer på tilstanden vises.

4.1.2.2 Visning av ferdigspilt parti og vinner

Det er relevant for publikum å vite når et parti er over og hvem som har vunnet et parti. Det er ikke alltid helt klart for alt publikum. I visningen av et sjakkparti viser vi hvem som spiller, representert med navn, over og under et sjakkbrett. Dette er informasjon hentet fra PGN. Chess.js har funksjonalitet for å avgjøre om et spill er ferdig, og om noen har vunnet. Dette blir tatt i bruk for å kunne gjengi denne informasjonen. Når et parti er ferdig, så blir ett brett mer gjennomsiktig, for å visuelt separerer det fra et pågående parti. Samtidig blir det presentert en tekst under partiet om at partiet er over og hvem vinneren var. Begge deler er illustrert i figur 6.

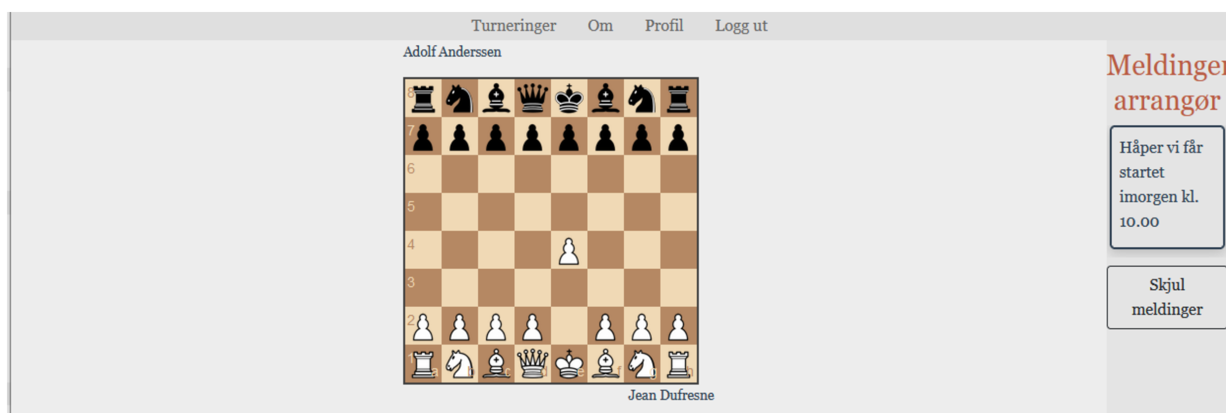


Figur 6 - Visning av ferdigspilt parti

4.1.2.3 Visning av en kamp

Turneringsskjermen kan vise alt fra ingen brett opptil flere. Aalesund Schaklag har 8 digitale sjakkbrett, så det maks antall brett skjermen er tilpasset visuelt for, hvor alt vises på en skjerm uten scrolling.

Figur 7 viser for en turnering hvor et parti spilles.



Figur 7 - Eksempel av turnering med et parti

4.1.2.4 Visning av flere kamper

Figur 8 viser en turnering hvor flere partier foregår.

Figur 8 - Eksempel av turnering med flere partier

4.1.2.5 Visning av meldinger

Den andre funksjonaliteten for turneringsskjermen er presentasjon av meldinger som er sendt av turneringsledelsen, her vist i figur 9. Slik som med ved sjakkpartier, så hentes disse meldingene regelmessig så bruker raskt for med seg eventuelle oppdatering fra turneringen.

Figur 9 - Eksempel av visning av meldinger sendt til turnering.

4.1.2.6 Visning av viktighet av melding

Ikke alle meldinger er like viktige. Derav ville gruppen synliggjøre en slik differanse. Turneringsledelsen har mulighet til å velge om en melding er viktig eller ikke. Dette gjengis med ulike farger på rammen av meldingene. En viktig melding får en oransje ramme, vist i figur 10, mens en ikke viktig melding har en svart ramme, vist i figur 11.

Figur 10 - Presentasjon av viktig melding

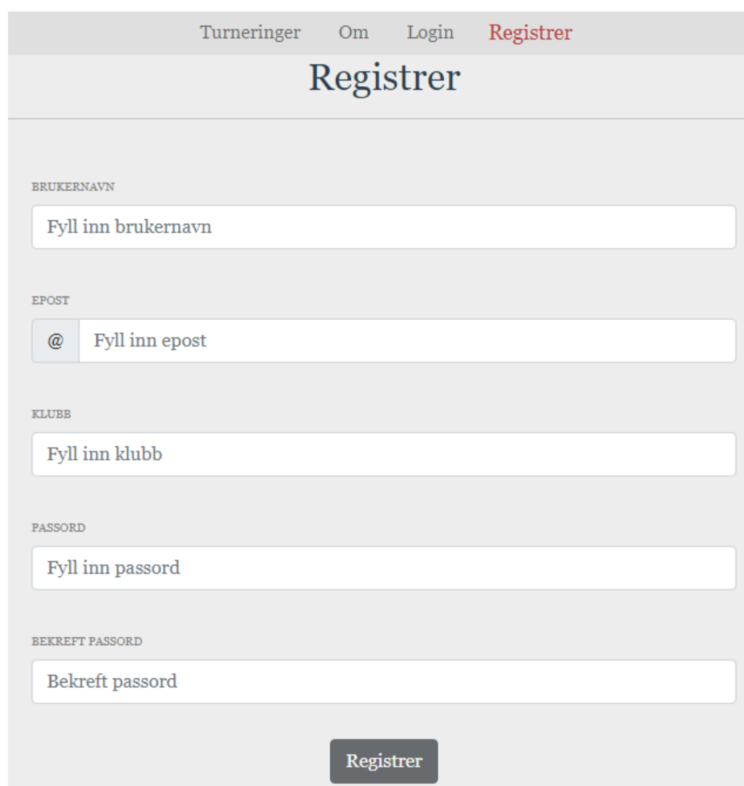
Figur 11 - Presentasjon av ikke viktig melding

4.1.3 Roller i systemet

Det var som mål at det skulle være mulig for alle som arrangerer turneringer med Turneringsservice og DGT LiveChess å ta i bruk løsningen. Det førte til at vi måtte innføre et brukerkonsept og roller. I systemet finnes det tre brukerroller. Det er administrator, organisator og vanlig bruker. En administrator har muligheten til å endre rollene til andre brukere, samt opprette turneringer. En organisator er en rolle tiltenkt de som organiserer turneringer. Disse har rettighetene til å opprette turneringer. Disse rollene som kan opprette turneringer blir så eiere av turneringen man oppretter. En bruker har per nå ingen funksjonalitet annen at det er en bruker som kan av administrator omgjøres til en organisator. Dette muliggjør at personer med riktig brukerrolle kan opprette turneringer, og hindrer at absolutt hvem som helst kan opprette turneringer.

4.1.4 Registrering

Når vi har et brukerkonsept, så må det være mulig å registrere seg i systemet.

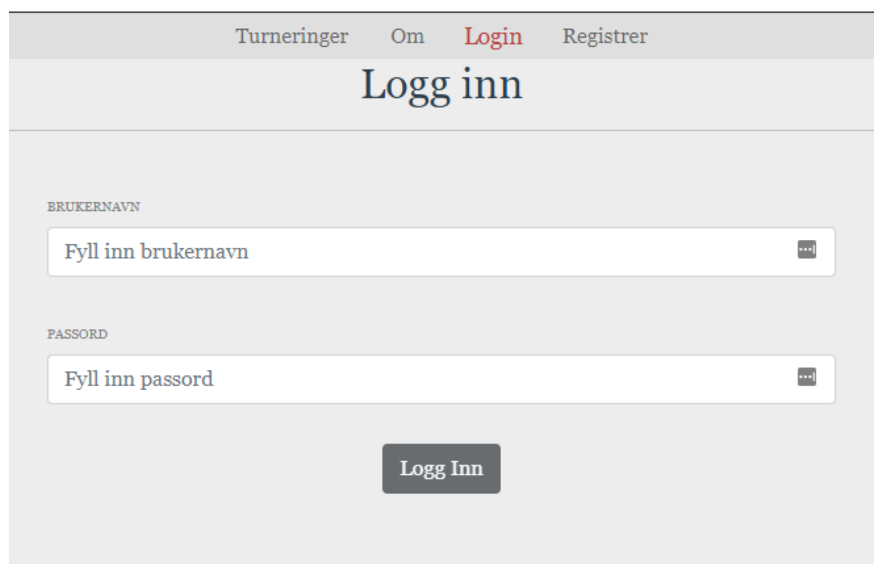


Figur 12 - Registreringsskjema.

Når en bruker ønsker å registrere seg kan de enkelt manøvrere seg til siden hvor de kan gjøre dette ved å trykke på "Registrer" knappen som er i navigasjonsbaren dersom brukeren ikke er autentisert. Når de trykker seg inn der blir de møtt av et skjema, figur 12, som inneholder de nødvendige tekstboksene de på fylle inn for å registrere seg. Her brukes valideringsbiblioteket Vuelidate til å implementere regler for gyldig input.

4.1.5 Login

Ette gyldig registrering blir så bruker blir automatisk rutet til denne skjermen etter gyldig registrering. Her blir de møtt av to tekstbokser de er nødt til å fylle inn. Bruker kan også navigere til login ved hjelp av navigasjonsbaren. Her brukes valideringsbiblioteket Vuelidate til å implementere regler for gyldig input. Figur 13 viser innlogginsskjemaet.



Figur 13 - Innloggingsskjema

4.1.6 Opprette turneringer

En innlogget bruker med riktig rolle, administrator eller arrangør, har muligheten til å opprette en turnering. Denne brukeren finner knappen tilgjengelig på oversikten over turneringer.

Figur 14 - Knapp for oppretting av turnering

Trykker brukeren på denne knappen, vist i figur 14, blir så bruker rutet videre til skjema for oppretting.

Figur 15 - Skjema for oppretting av turnering

Når en turnering skal bli opprettet kreves det spesifikke informasjoner som skal måtte bli fylt ut, vist i figur 15. Her brukes valideringsbiblioteket Vuelidate til å implementere regler for gyldig input. Om oppretting går gjennom, vil brukeren bli navigert videre til et turnerings-dashboard.

4.1.7 Dashboard

Dashboard har som hensikt å gi turneringsledelsen et område å styre en turnering, i vårt system, fra. Dashbordet, vist i figur 16, gir eieren av turneringen muligheten til å se tidligere sendte meldinger, sende meldinger, angi viktigheten av meldingen, slette sendte meldinger og slette turneringene.

Figur 16 - Visning av Dashboard for turnering

4.1.8 Sende meldinger

På Dashbordet kan eieren av turneringen sende meldinger. Det er implementert med en enkel meldingsfelt, viktighetsfelt og en knapp for å sende. Begge feltene må være utfylt for å kunne sende meldingen. Denne funksjonaliteten er vist i figur 17 og figur 18, hvor knappen grået ut og ikke kan interakteres med.

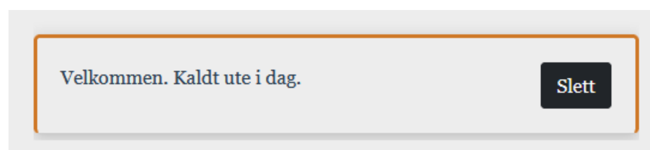
Figur 17 - Eksempel av deaktivert send-knapp

Figur 18 - Eksempel av aktivert send-knapp.

Her brukes valideringsbiblioteket Vuelidate til å implementere regler for gyldig input.

4.1.9 Slette meldinger

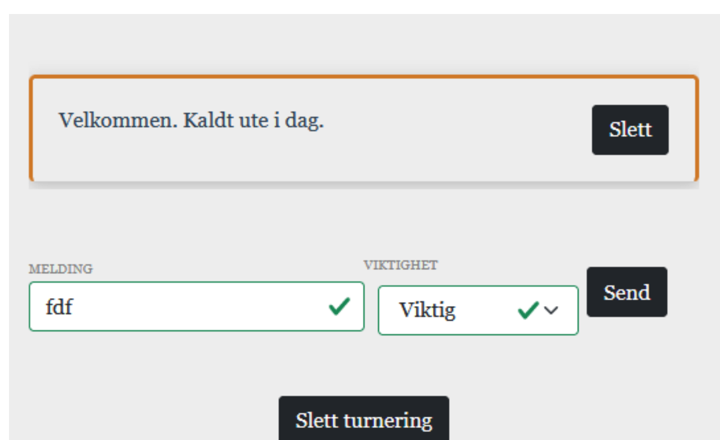
Det kan hende en melding inneholder en skrivefeil eller feilaktig informasjon. Eieren av turneringen har så mulighet til å slette en melding. Figur 19 viser en melding med en slett-knapp.



Figur 19 - Slett-knapp plassert til høgre i melding.

4.1.10 Slette turnering

Vi valgte å gi en bruker muligheten til å slette en turnering. Når man sletter en turnering, vil meldinger og sjakkpartier knyttet til denne også turneringen slettes. Man trykker på Slett turnering-knappen, vist i figur 20, og blir så navigert tilbake til listen over turneringer.



Figur 20 - Slett-knapp for turnering plassert nederst i Dashboard.

4.1.11 Profil

Når systemet har et brukerkonsept, følte vi det var naturlig å ha en slags profil. Denne er enkelt implementert med å presentere informasjon om brukeren, samt alle turneringen brukeren er eier av i systemet. Dette gir samtidig en bruker raskere navigering til sine egne turneringer. Et eksempel av en brukers detaljer og tilknyttede turneringer er vist i figur 21.

Figur 21 - Visning av brukerprofil

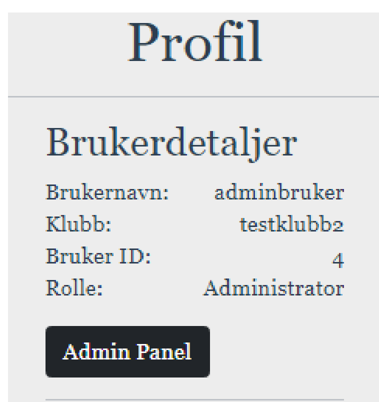
4.1.12 Administrator-panel

Administrator-panelet er et panel bare for brukere som er administrator. I dette panelet har brukeren muligheten til å endre rollene til andre brukere enn seg selv, hvor skjema er vist i figur 22.

Figur 22 - Visning av Administrator-panel

Dette panelet er tilgjengelig gjennom profilen, figur 23 og 24 viser at knappen for å nå Administratorpanelet bare er tilgjengelig for en bruker med Administrator-rollen.

Figur 23 - Brukerdetaljer - andre brukere



Figur 24 - Brukerdetaljer - Administrator

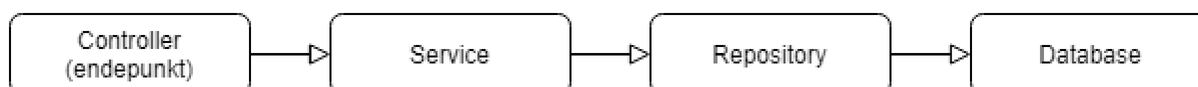
4.2 Back-end

Backend er implementert med en lagvis-arkitektur med et REST-API. Videre underkapitler vil gå dypere inn i dette og gi en detaljert eksempel av implementasjonen av en lagvis arkitektur.

4.2.1 Lagvis arkitektur

Backenden er implementert som en lagvis-arkitektur. Den definerer endepunkter for meldinger, turneringer, sjakkpartier og autentisering. Hvert av disse delen har presentasjonslaget, forettningslogikklaget, persistenslaget og databaselaget. Der tjenester som krever autorisering blir et sikkerhetslag lagt på.

En slik arkitektur i en Spring kontekst blir seende slik ut, som vist i figur 25.



Figur 25 - Illustrasjon av lagvis arkitektur implementert med Spring annoteringsnavn.

Spring rammeverket og JPA innfører en del annoteringer, som lar man definere hvilken funksjon ulike Beans, eller klasser som Spring styrer, skal ha. Disse er Controller eller RestController, Service, Repository og Entity.

Controller/Restcontroller

Controllerlaget kan både sees på som både å være en del av presentasjonslaget og forretningslogikk, i og med de definerer grensesnittet ut mot klienter, samt inneholder logikk for hvordan man skal behandle en innkommende forespørsel.
I vårt tilfelle vil klasser i dette laget inneholde:

- Hva slags metoder de skal ta imot, POST/GET o.l.
- Hva endepunktet sin sti skal være, "api/auth/login".
- Hva slags autorisering er nødvendig, om noen. F.eks. kun for Administrator autoriserte.
- Kall til serviceklassen for videre håndtering
    - Kan også inneholde andre prosesser før kall til serviceklassen avhengig av kallets nødvendighet

Service
Service svarer til forretningslogikklaget. Hos oss inneholder dette laget:
- Hva skal bli gjort med kallet?
- I gruppens applikasjon er det i hovedsak databasehandlinger som blir gjort slik at i de fleste metodene i servicelagene blir det gjort videre kall til repositorylaget.

Repository
Repository er en del av persistenslaget. Repository annoteringen sier at denne Bean-en er ansvarlig for å interaktere med databaselaget. Oppgavene dens er:
- Håndtering av databasehandlinger.
- Ettersom vi benytter Spring Boot med Spring Data JPA som implementerer JPA er det ofte veldig lite som er definert i disse klassene. Årsaken er som nevnt i teoretisk grunnlag delen av Spring Data JPA inneholder mange forhåndsdefinerte handlinger slik at man ikke nødvendigvis trenger å legge til noen egendefinerte metoder.
- Eventuelle egendefinerte metoder vil bli definert her, enten i form av enklere som bygger videre på JPA eller fullstendige egendefinerte med konkrete databasekall.

Entity
Entity blir brukt for å merke at en klasse tilsvarer en entitet eller domeneobjekt. Dette er en nødvendig annotering at ORM-er, slik som Hibernate, skal kunne mappe klassen til en tabell i relasjonsdatabase. Det er med andre ord en del av persistenslaget.
Modell klassene inneholder:
- Deklarering av variabler med datatyper
- Konstruktører, flere avhengig av behov.
- Getters og setters for å håndtere henting og endring av data til objekter.

4.2.2 REST

Backenden ble implementert med REST-endepunkter hvor presentasjonslaget kan hente informasjon fra. Disse REST-endepunktene tar imot og returnerer i filformatet JSON.

4.2.2.1 Implementerte REST-endepunkter

Ikke-autentiserte:

- ❑ Registrere
  POST /api/auth/register
- ❑ Logge inn
  POST /api/auth/login

Ikke-autentisert og alle brukere

- ❑ Se turneringer
  GET /api/tournament
- ❑ Se partier i turnering
  GET /api/games
- ❑ Få oppdaterte aktive partier
  PUT /api/games/updategame?gameid={gameId}
- ❑ Se meldinger til turnering
  GET /api/message/specific?tournamentId={tournamentId}

Autentiserte brukere med arrangør eller administrator rolle:

- ❑ Lage turnering
  POST /api/tournaments/createtournament
- ❑ Se sine turneringer
  GET /api/tournaments/tournamentsbyowner?{ownerId}
- ❑ Slette sine turneringer
  DELETE /api/tournaments/delete?{tournamentId}
- ❑ Sende meldinger til sine turneringer
  POST /api/message/
- ❑ Slette meldinger til sine turneringer
  DELETE /api/message/delete?messageId={messageId}

- ❑ Legge til partier til turnering
  POST /api/tournaments/addgame
- ❑ Sette antall partier per runde
  POST /api/tournaments/setgamesperround
- ❑ Sette antall runder
  POST /api/tournaments/setnumberofrounds

Autentiserte brukere med administrator rolle:

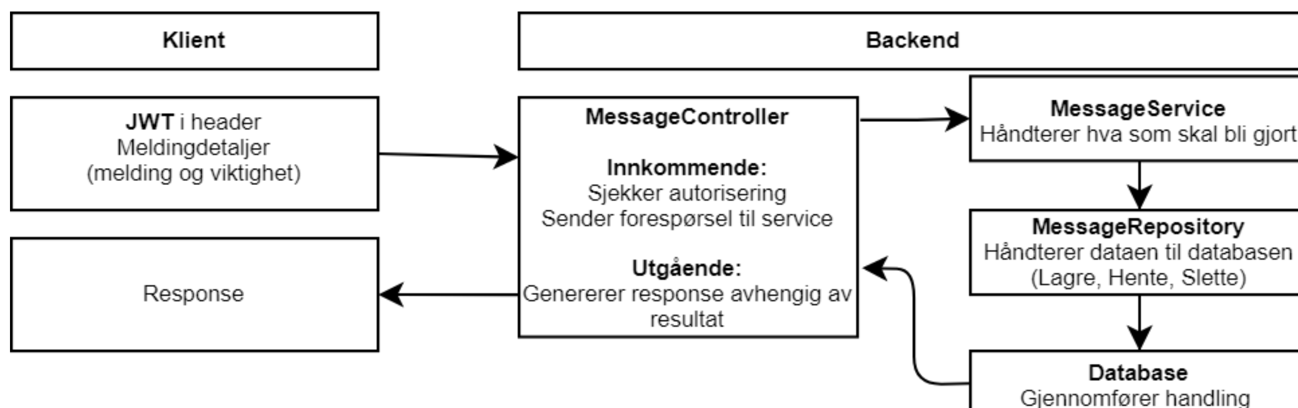
- ❑ Hente brukerdetaljer
  GET /api/auth/getUserDetails?userId={userId}
- ❑ Endre rollen til brukere
  PUT /api/auth/updateRole?userId={userId}&roleId={roleId}

4.2.3 Detaljert gjennomgang av implementering av lagvis arkitektur

Tjenestene for melding, turnering og livesjakk er laget med den samme lagvis arkitekturen. I neste punkt vil det bli beskrevet hvordan dette er løst, med meldingstjenesten brukt som eksempel.

4.2.3.1 Eksempel – Meldingstjeneste

En illustrasjon av informasjonsflyten for meldingstjenesten gjennom en lagvis arkitektur er vist i figur 26.



Figur 26 - Informasjonsflyt for meldingstjeneste

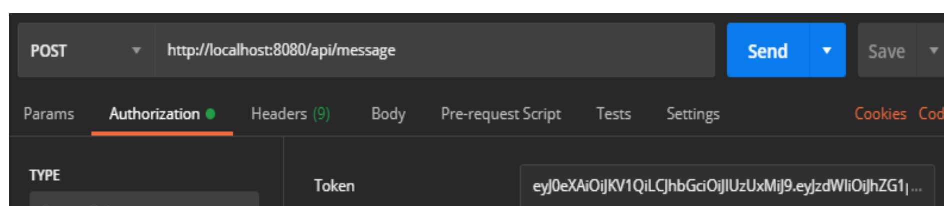
En arrangør skal kunne gjøre diverse tjenester relatert til meldinger på sine turneringer for å kunne informere tilskuere. Dette har vi løst ved å opprette støtte for dette i APIen på backenden ved å lage diverse endepunkter. En arrangør vil kunne gjennomføre følgende tjenester relatert til meldinger:

- Sende meldinger
- Slette meldinger

I tillegg så skal alle tilskuere kunne se meldingene.

Sende meldinger

Når en arrangør ønsker å sende en melding til en av sine turneringer kan dette gjøres på følgende måte via en POST forespørsel til backenden. Først og fremst må brukeren være innlogget og autentisert. Når en bruker logger inn får den i respons et JWT token, dette er viktig og nødvendig videre. Denne må legges i headeren av forespørselen. I figur 27 viser man hvordan man gjør dette i Postman.

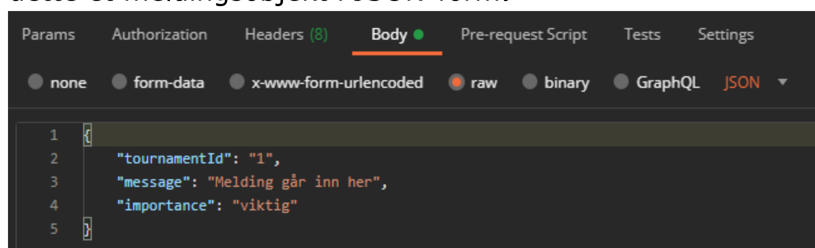


Figur 27 - Eksempel av at man legger til JWT i header til en forespørsel

Deretter må det fylles ut informasjonen som skal bli sendt og lagret i databasen. Dette gjøres i form av et JSON-objekt, som vil være i bodyen til forespørselen. I dette JSON-objektet må det fylles inn ønsket data, for en melding er det tre detaljer en ønsker.

- TurneringsID - Hvilken turnering den gjelder
- Melding - Selve meldingen
- Viktighet - Om den er viktig eller ikke.

Figur 28 viser dette et meldingsobjekt i JSON-form.



Figur 28 - Eksempel av meldingsobjekt i JSON-form vist i Postman

Dette blir da sendt videre til backenden. Første punktet forespørselen møter er endepunktet, som er deklarert i **MessageController**. Denne klassen er definert med notasjonen *@RestController*.

MessageController

```java
@PostMapping
@CrossOrigin(origins =  "*", allowedHeaders = "*")
@PreAuthorize("hasAuthority('ROLE_ORGANIZER')")
public void sendMessage(@RequestBody MessageRequest message) {
    UserDetailsImpl userDetails = (UserDetailsImpl) SecurityContextHolder.getContext().getAuthentication()
            .getPrincipal();
    Long userId = userDetails.getUserId();
    Tournament tournament = tournamentService.getTournament(message.getTournamentId());
    if (!tournament.getOwner().getUserId().equals(userId)) {
        throw new NotAuthorizedException();
    }
    messageService.sendMessage(message, tournament);
}
```

Figur 29 - Implementasjon av sendMessage-metoden i MessageController

I koden, vist i figur 28, kan vi se et par annoteringer

- **@PostMapping**
  - Annotasjon som er en spesifisering av @RequestMapping notasjonen som definerer at metoden håndterer POST forespørsler som treffer endepunktet.
- **@PreAuthorize("hasAutority('ROLE_ORGANIZER')")**
  - Definerer hva slags autorisering som er nødvendig for denne metoden.
  - Autorisering er beskrevet i mer detalj i punkt om Autorisering.
  - I dette tilfellet er du nødt til å være enten arrangør eller administrator.
- **@CrossOrigin** er en notasjon som man må ha enten globalt for klassen, i individuelle metoder eller i noen tilfeller begge for å tillate forespørsler fra visse kilder. Dersom dette mangler vil ikke forespørselen bli akseptert i det hele tatt da man vil støte på CORS-error.

Videre inne i metoden er det flere hendelser. Først blir det opprettet et objekt kalt **userDetails** som implementerer **userDetailsImpl**. Som er måten en i backenden kan hente ut informasjon om brukeren som gjennomførte forespørselen. Videre henter vi ut IDen til brukeren. Dette gjøres for å verifisere at denne brukeren er eieren av turneringen for å unngå at brukere kan legge til meldinger i andre sine turneringer.

Dersom brukeren ikke er eier av turneringen vil det bli kastet et exception, en feilmelding og forespørselen blir avbrutt og nektet. Dersom brukeren er eieren av turneringen til forespørselen bli sendt videre til **MessageService** for videre håndtering.

MessageService

MessageService er klassen som inneholder hva som skal bli gjort med forespørselen, typisk sett i fleste av våre metoder og i alle meldingstjenester gjelder dette handlinger mot databasen.

Videre går vi med eksempelet om sending av meldinger fra arrangør. Nedenfor, i figur 30, er metoden som blir kalt av metoden i **MessageController.**

```
public void sendMessage(MessageRequest message, Tournament tournament) {
    Message messageEntity = new Message();
    messageEntity.setTournament(tournament);
    messageEntity.setDate(LocalDate.now());
    messageEntity.setMessage(message.getMessage());
    messageEntity.setImportance(message.getImportance());
    messageRepository.save(messageEntity);
}
```

Figur 30 - Implementasjon av sendMessage-metoden i MessageService

Her ser vi at den har to parameter, **MessageRequest** og **Tournament**. Den førstnevnte er en modellklasse for melding som inneholder informasjon om variabler og ønsket data. I metoden blir det opprettet et objekt av dette modellen. Og informasjonen i forespørselen blir satt til dette objektet. I tillegg er det en variabel vi ikke definerte i JSON-objektet, som er datoen. Datoen blir satt i når en melding skal bli lagret i databasen, den er ikke egendefinert. **Tournament** inneholder tuneringsIDen og knytter en melding og en turnering sammen. Videre blir dette objektet sendt videre til **MessageRepository**.

MessageRepository

MessageRepository er en Interface-klasse som håndterer alt av hendelser som skal gjøres med databasen. Den bygger videre på **JPARepository**, som er en implementasjon av JPA. Spring Data JPA er bygd opp på en slik måte at de har mange forhåndsdefinerte metoder for databasehandlinger. Noe som gjør at selve Interface-klassen ser merkverdig tom ut, som vist i figur 31.

```
@Repository
public interface MessageRepository extends JpaRepository<Message, Long> {

}
```

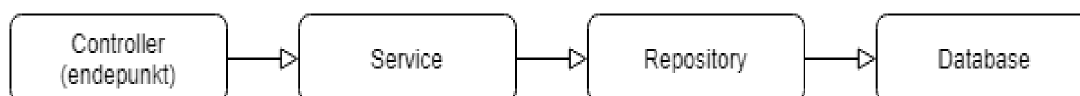
Figur 31 - Implementasjon av MessageRepository

Eneste som er definert her er at i defineringen av Interface-klassen bygger vi videre på JPARepository som nevnt ovenfor, og der definerer vi to parameter. Første, **Message** er hvilket objekt det skal være, og **Long** er datatypen til identifiseringen av objektene.

I kallet i **MessageService** ser vi at metoden som blir kalt er *messageRepository.save();*. Som vi kan se at ikke eksiterer i Interface-klassen, dette er fordi at som nevnt er det mange forhåndsdefinerte metoder for håndtering av data til databasen, og *save()* er en av disse. Når denne metoden da blir kalt til meldingen bli lagret i databasen.

Se meldinger

Tilskuere skal kunne se meldinger som er relatert til turneringen de ser på. Måten dette er løst er via et kall mot APIen i backenden. I likhet med å sende meldinger er arkitekturen satt opp på samme måten med kommunikasjon på tvers av de diverse lagene, henvist til i figur 32.



Figur 32 - Illustrasjon av lagvis arkitektur

```java
@GetMapping
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RequestMapping(path = "/specific")
public List<Message> getMessageTournament(@RequestParam Long tournamentId) {
    return messageService.getMessageTournament(tournamentId);
}
```

Figur 33 - Implementasjon av getMessageTournament-metode i MessageController

I figur 33 viser man metoden getMessageTournament i **MessageController**. Notasjonene her er litt annerledes enn på å sende meldinger, men har et par likheter.

- **@GetMapping** - Notasjon om at den skal ta imot GET forespørsler.
- **@CrossOrigin** - Notasjon for akseptering av kilde av forespørsel.
- **@RequestMapping** - Notasjon for definering av stien endepunktet skal ligge bak.

Selve metoden er bygd opp som en liste da vi ønsker å få returnert en liste over alle meldinger tilhørende turneringen. Turneringen det er ønsket å få meldingene for spesifiseres som en parameter. Denne parameteren blir mottatt som et RequestParam noe som blir definert med notasjonen **@RequestParam.** Videre her blir det kalt en metode inne i **MessageService**, som er vist i figur 34.

```java
public List<Message> getMessageTournament(Long tournamentId) {
    return messageRepository.findAll()
            .stream()
            .filter(c -> tournamentId.equals(c.getTournament().getId()))
            .collect(Collectors.toList());
}
```

Figur 34 - Implementasjon av getMessageTournament-metode i MessageService

Denne metoden returnerer en filtrert liste. Denne filtrerte listen gjøres ved at det først blir gjort et kall til **messageRepository** som har en forhåndsdefinert metode, findAll***().*** Denne henter en liste over meldingene. Deretter blir det benyttet en funksjon i Java om å strømme dataen, underveis i dette filtrerer vi listen og henter ut meldingene som har

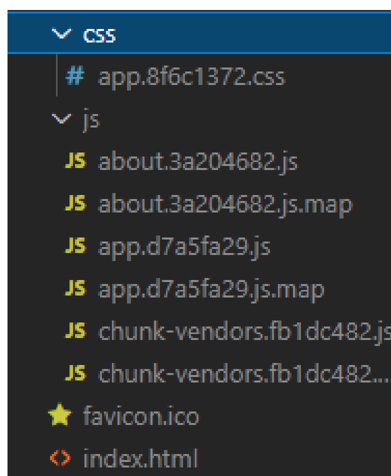
tilsvarende turneringsID som vi har sendt med som en parameter. Disse blir da hentet inn og returnert inn i listen. Listen blir da returnert som respons til forespørselen.

4.3 Frontend

Frontend er gjennomført som en Single Page Application, med Client Side Routing ved hjelp av presentasjonslag-rammeverket Vue.js, samt flere andre rammeverk og biblioteker. Videre underkapitler vil gå dypere inn i dette.

4.3.1 Single Page Application

Frontenden, eller presentasjonslaget, ble utformet som en Single Page Application. Begrunnelsen var å kunne gi sluttbrukeren en myk opplevelse på siden, der man ikke oppdaterer siden. Samtidig ville tjenesten ha innhold som må oppdateres ofte og med unikt innhold. Dette ville vært vanskelig å implementere med Server Side Rendering, hvor dette innholdet må bli generert på backend.



Figur 35 - Innholdet i bygg-mappen. Viser bare en HTML-fil

Når man bygger prosjektet er det disse filene som blir generert, hvor man ser at det den eneste HTML-filen er index.html, altså den eneste siden eller eneste "Page". Dette er vist i figur 35.

4.3.2 Client Side Routing

Siden dette er en Single Page Application må det gjøres ruting på presentasjonslaget. I praksis ble dette utført med Vue Router. Får å rute innen applikasjonen må man oppfylle to krav. Det ene er å definere en rute. Ruten må si hvilken sti innenfor applikasjonen den skal tilsvare, hvilket navn skal bruke for å referere til ruten, hvilken komponent som skal lastes inn når man går til den ruten. Stien man definerer blir gjengitt i URL i nettleser. Implementasjon av en rute er vist i figur 36.

```
const routes = [
  {
    path: "/",
    name: "Tournaments",
    component: Tournaments,
    meta: {
      requiredAuth: false
    }
  },
```

Figur 36 - Eksempel av definisjonen av en rute.

Det andre man kravet man må oppfylle er å pakke inn et HTML-element med et Vue.js element kalt router-link hvor man sier til hvilken sti man skal gå til. HTML-elementet oppfører seg da som en hyperlenke. Figur 37 viser hvor lett dette gjøres.

```
<h1>Vi fant dessverre ikke det du lette etter.</h1>
<router-link :to="{ name: 'Tournaments' }">
  <p>Naviger tilbake til fremsiden.</p>
</router-link>
```

Figur 37 - Eksempel av Router-link rundt HTML-element for navigering

4.3.3 Single File Components

Alle komponentene i applikasjonen ble implementert som Single File Components. Disse komponentene lar man definere HTML (gjennom Template Syntax), JavaScript og CSS i en og samme komponent. Det som er veldig hendig er at man kan anvise at CSS skal bare treffe denne komponenten, gjennom å angi at <style scoped>, som gjør at man kan gjenbruke navn for klasse- og id-attributter i forskjellige komponenter samt unngå at CSS påvirkes i andre komponenter. Strukturen av en Single File Component er vist i figur 38.

```
<template>
  <div v-if="hasTournaments" class="tournament-list">
    <h3>
      {{ title }}
    </h3>
    <TournamentItem class="tournament-item"
      v-for="tournament in tournaments"
      :key="tournament"
      :tournament="tournament"
    />
  </div>
</template>

<script>···

<style scoped>···
```

Figur 38 - Eksempel av strukturen av en Single File Component

4.3.4 Vuex og tilstandsbehandling

I vår løsning ble det nødvendig med en global tilstandsbehandling, først og fremst for å håndtere autorisering og håndtere brukerkonseptet. Det var flere komponenter som måtte gjøre kall til backend som krevde autorisering med JWT. Alle disse komponentene må ha tilgang på JWT, samt være sikker på denne er den samme for alle komponenter. Vuex forenklet for oss den nødvendige logikken for å håndtere dette. Det ble enda et konsept og bibliotek å lære, men det ble en god løsning å bruke Vuex.

4.3.5 Inputvalidering

I løsningen ble det en del input felt. Det krever da at man gir validering en tanke, for å hindre at ugyldige verdier og formater blir gitt. Gruppen tok i bruk biblioteket Vuelidate til å hjelpe med dette. Vuelidate krever at man innfører en funksjonsegenskap i dataobjektet til Vue komponenten, referert til som v$, her vist i figur 39. Det er da denne funksjonen som tar av seg selve valideringen. Samtidig må komponenten definere noe data å kunne validere, her vist med username og password.

```
data() {
  return {
    v$: useValidate(),
    username:'',
    password:''
  }
```

Figur 39 - Eksempel av bruk av Vuelidate funksjon

Man refererer så til denne funksjonen i Template for komponenten, som vist i figur 40.

```
<div v-if="v$.username.
```

Figur 40 - Eksempel av referanse til Vuelidate-property i Template Syntax

Det siste man trenger å gjøre er å definere regler som skal gjelde for dataen man skal validere. Vuelidate kommer med forhåndsdefinerte regler kan si data i en komponent skal ha. Figur 41 viser to av 3 av de mulige reglene, at en input må oppgis, at det ene skal være alfanumerisk samt ha en minimumslengde av 1.

```
validations() {
  return {
    username: {
      required,
      alphaNum,
      minLength: minLength(1)
    },
    password: {
      required
    }
  }
```

Figur 41 - Eksempel av definering av Vuelidate regler.

4.3.6 Axios - HTTP klient

For å sende kall til backend-en er man nødt til å bruke en slags HTTP klient, gruppen valgte å benytte Axios til dette. Axios tilbyr enkelt å bygge opp klient for å sende kall.

Gruppen definerte noen service-filer hvor Axios klienten lå. Dette lot man unngå at man oppretter en klient for hver komponent. Oppretting er vist i figur 42.

```
const apiClient = axios.create({
  baseURL: backendURL,
  withCredentials: false,
  headers: {
    Accept: "application/json",
    "Content-Type": "application/json"
  }
});
```

Figur 42 - Eksempel av oppretting av Axios-instans

I samme service fil definerte vi metodene som skulle eksponeres til komponentene, her illustrert med en funksjon, i figur 43.

```
getGames(tournamentId) {
  return apiClient.get(gamesByTournament, {
    params: { tournamentid: tournamentId }
  });
```

Figur 43 - Eksempel av en funksjon i en service-fil.

Axios støtter asynkrone funksjoner. Som gjør logikken for håndtering av kall til backend mye enklere. Figur 44 viser syntax for håndtering av respons ved slike asynkrone kall.

```
GameService.getGames(this.tournamentId)
  .then(response => {…
  .catch(error => {…
```

Figur 44 - Eksempel av syntax for håndtering av respons ved asynkrone kall.

4.3.7 Utseende

Utseende til applikasjonen ble skrive i ren CSS og CSS-rammeverket Bootstrap. Dette gir applikasjonen et noe eldre utseende. Begrunnelse for valget av bruk av CSS og Bootstrap, blir diskutert i drøfting.

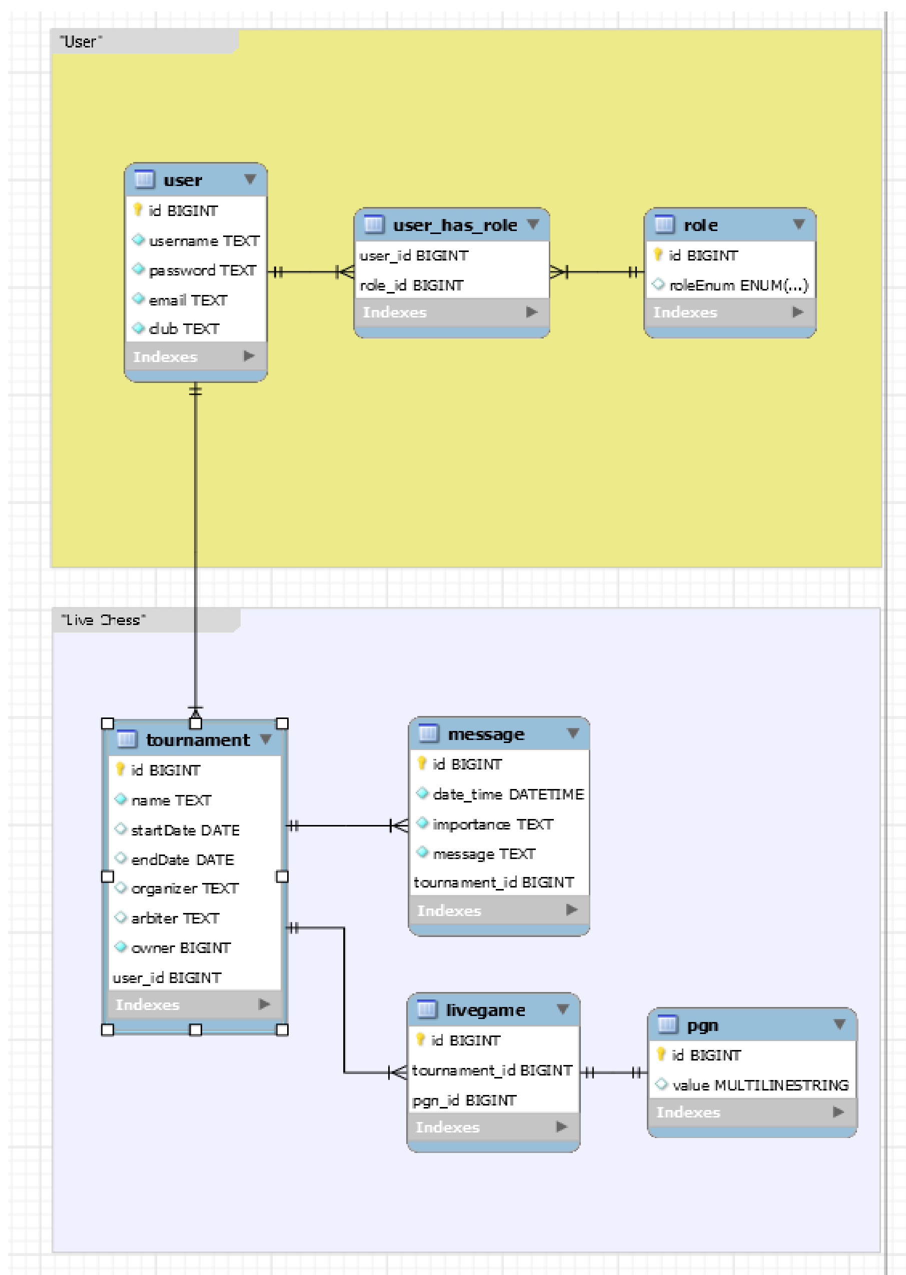
4.4 Database

Herunder vil vise hvordan Database-laget ble implementert gjennom design, tabeller, forhold og befolking av testdata.

Valget av databasesystem falt på PostgreSQL, men på grunn av god abstraksjon i Spring rammeverket mot databaselaget, kan man bruke omtrent hvilken som helst kjent databaseleverandør.

4.4.1 Design

Designet av databasen består av samlinger. Det som er knyttet til brukere og det som er knyttet visningen av sjakkpartier. Det ble totalt 7 tabeller, som vist i figur 45. Databasedesign er også lagt til som vedlegg L.



Figur 45 - Diagram for databasedesign

4.4.2 Tabeller

Dette delkapittelet går gjennom alle tabellene som er definert for databasen.

Tabelloppsett

Gruppen har, som sett i designet ovenfor, satt opp syv tabeller. Hver eneste entitet er representert av en egen tabell og relasjoner er satt opp basert på hva som trenger å tilhøre hverandre.

User-tabell

Tabellen for brukere er satt opp for å inneholde den nødvendige og ønskede informasjonen som er ønsket å vite om en gitt bruker

- ID - *Primary Key* - 32 bit størrelse
- Username – Varchar, opp til 25 tegn i størrelse.
- Password – Varchar, opp til 75 tegn i størrelse. Blir kryptert før den blir lagt inn i databasen.
- Epost – Varchar, opp til 50 tegn i størrelse
- Klubb - Varchar.

Role-tabell

Rolle tabellen blir generert basert på modellklassen dens og en Enum klasse som definerer hvilke roller som skal bli generert. Basert på oppsettet gruppen har valgt for roller blir det automatisk laget tre grupper.

- ID - 32 bit størrelse
- roleEnum - ENUM/Varchar.

User_role-tabell

Dette tabellen blir brukt for å beskrive forholdet mellom en bruker og rollen den har. I denne kan man enkelt se hvilken rolle en gitt bruker har.

- user_id - *Foreign key* - 32 bit størrelse
- role_id - *Foreign key* - 32 bit størrelse

Tournament-tabell

Tournament tabellen inneholder informasjonen om turneringene

- ID - *Primary Key* - 32 bit størrelse
- Name - Varchar
- StartDate - Dato
- EndDate - Dato
- Organizer - Varchar
- Arbiter - Varchar
- Owner - *Foreign Key* - 32 bit størrelse.

LiveGame-tabell

Inneholder informasjonen og linker sammen PGN inn til partier og linker de igjen til en turnering. Skal representere et sjakkparti.

- ID - *Primary Key* - 32 bit størrelse
- tournament_id - *Foreign Key* - 32 bit størrelse
- pgn_id - *Foreign Key* - 32 bit størrelse

PGN-tabell

Inneholder PGN for et parti.

- ID - *Primary key* - 32 bit størrelse
- Value - Varchar.

Message-tabell

Inneholder informasjonen om meldingene til en turnering

- ID - *Primary Key* - 32 bit størrelse.
- Date_time - Dato
- Importance - ENUM/Varchar

- Message - Varchar
- Tournament_ID - *Foreign Key* - 32 bit størrelse.

4.4.3 Forhold

Måten gruppen valgte å løse autorisering var ved roller som brukere kunne ha. Det ble beskrive et enveis mange-til-mange forhold mellom tabellene User og Role ved hjelp av JPA annoteringer i entitetsklassen User. Valget falt på et mange-til-mange da vi så for oss det skal være mulig for en bruker å ha flere roller. Siden dette er et mange-til-mange forhold vil Hibernate opprette hjelpetabellen User_Role, som vist i figur 46.

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "user_role", joinColumns = @JoinColumn(name = "user_userId"),
        inverseJoinColumns = @JoinColumn(name = "role_roleId"))
private Set<Role> roleSet = new HashSet<>();
```

Figur 46 - Eksempel av definisjon av et mange-til-mange forhold

Videre var oppsettet for en turnering slik at under opprettelsen av en turnering blir brukeren som opprettet den satt til eieren. Her ble det da nødvendig med et et-til-mange forhold mellom bruker- og turneringstabellene. Dette er fordi en bruker skal kunne ha flere turneringer den er eieren av. Her ble da bruker tabellen sin primærnøkkel, altså bruker IDen hentet inn som sekundær nøkkel og satt som et attributt til en turnering.

Meldinger ble også knyttet opp mot turneringen den handlet om. Her ble det knyttet et one-to-many forhold da en turnering skal kunne ha flere meldinger relatert til en. Her ble det igjen knyttet et forhold med IDene som primær/sekundærnøkler.

Til slutt er også sjakkpartiene, representert med tabellen LiveGame, knyttet til turneringstabellen.

Disse forholdene knyttet til turneringstabellen muliggjør det å slette all tilhørende informasjon som hører til en turnering, slik som meldinger og sjakkpartier. Dette gjøres med å innføre Cascade-regler, hvor en operasjon utført på turneringstabellen også bli utført på tabellene i dette forholdet.

```java
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
        mappedBy = "tournament")
@JsonIgnore
private List<LiveGame> games;

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
        mappedBy = "tournament")
@JsonIgnore
private Set<Message> messages;
```

Figur 47 - Eksempel av definisjon av et et-til-mange-forhold

Det siste forholdet å beskrive er mellom LiveGame og PGN. Dette er et et-til-et-forhold. Logikken er at en PGN skal bare høre til et sjakkparti, noe som stemmer overens med hva PGN representerer. Figur 48 og figur 49 viser hvordan dette er gjort for LiveGame og PGN.

```java
@OneToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "livegame_id", nullable = false)
private LiveGame liveGame;
```

Figur 48 - Eksempel av ene siden av et tosidig et-til-et-forhold

```java
@OneToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "livegame_id", nullable = false)
private LiveGame liveGame;
```

Figur 49 - Eksempel av den andre siden av et tosidig et-til-et-forhold

4.4.4 Befolke med testdata

Under oppstart av backend serveren blir databasen kjørt opp og generert av seg selv via. Hibernate og Spring Data JPA. Via Spring kan man enkelt kjøre opp en database basert på innstillingene man spesifiserer i de diverse klassene. Under utvikling i lokalmiljøer var oppførselen til denne slik at den var satt til *create-drop*, som sletter hele databasen og setter opp igjen fra bunnen av. Dette var fordi gruppen benyttet CommandLineRunner metoder for å legge inn "dummy data" i databasen, og dette ga noen ganger problemer uten *create-drop* da den kunne finne duplikater som gjorde at serveren ikke kom opp. Utenom dette ble det brukt *validate* som kun kjører gjennom og sjekker om alt er som det skal.

4.5 Sikkerhet

Dette delkapittelet tar for seg hvilke sikkerhetstiltak som ble gjort for løsningen, samt hvordan de ble implementert.

4.5.1 HTTPS

Siden tjenestene operere med brukernavn og passord, og en frontend og en backend som kan kjøre på forskjellige servere, er det nødvendig med HTTPS på frontend og mellom frontend og backend.

HTTPS Sikring av frontend ble gjort med å bruke en web-hosting-tjeneste kalt Render.com. De tilbyr gratis hosting for statiske filer, slik som vår frontend, og tilbyr HTTPS som standard. I tillegg kan man kjøre opp siden direkte fra GitHub repository.

Dette er derimot ikke nok. Siden frontend har HTTPS inkludert, så kan fortsatt brukernavn og passord bli sendt i klartekst mellom frontend og backend. Samtidig er moderne nettlesere såpass strenge på en sluttbrukers vegne at de ikke tillater at det gjøres HTTP kall fra tjenesten om presentasjonslaget har HTTPS aktivert.

Backend måtte så sikres med HTTPS. Dette ble gjort med å skaffe oss et SSL-sertifikat fra en verifisert leverandør, SSLforFree.com. De tilbyr gratis 90-dagers sertifikater, som er nok for presentasjon av vår løsning. Samtidig vår Spring backend starte opp i Secure-modus og ha dette sertifikatet. Dette krever at vi i application.properties filen, Spring sitt eksterne API for konfigurasjon for applikasjonen, definerer blant annet at SSL skal være krevd og hvilken port man skal bruke. Dette er vist i figur 50.

```
server.port=8443
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:sjakkskjerm-PKCS-12.p12
server.ssl.key-store-password=hemmeligpassord
server.ssl.key-alias=navnet til sertifikatet
security.require-ssl=true
```

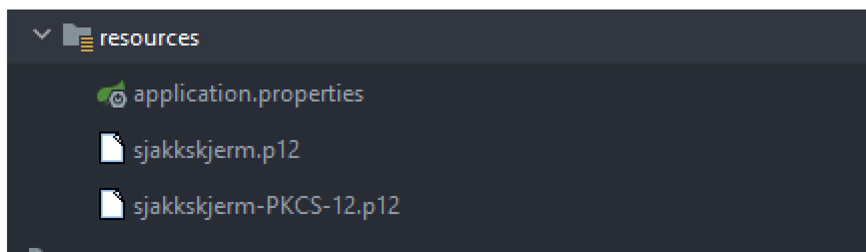
Figur 50 - Eksempel av nødvendig konfigurasjon for HTTPS i Spring-rammeverket.

I tillegg må Spring backenden konfigureres med at den fra oppstart skal kreve at alle innkommende forespørsler skjer over HTTPS, her vist i figur 51.

```
http
    .requiresChannel()
    .anyRequest()
    .requiresSecure();
```

Figur 51 - Eksempel av kode som kreves for at HTTPS skal bli aktivert ved oppstart av server.

Til slutt må sertifikatet legges i resources-mappen for Maven-prosjektet, og pakkes sammen med applikasjonen, her illustrert i figur 52.



Figur 52 – Eksempel av plassering av sertifikat.

4.5.2 JWT

Som nevnt igjennom rapporten, har JWT blitt brukt for å forenkle autorisering for operasjoner som bare skal kunne gjøres av rollene administrator og organisator, så brukernavn og passord ikke trenger å bli sendt ut til frontend for å igjen bruke på backend.

4.6 Detaljert gjennomgang av implementasjon av brukertjenester.

Dette kapittelet har som hensikt å vise hvordan flere av verktøyene og teknologiene ble brukt for å løse utfordringer og gi oss nødvendig funksjonalitet. Dette tar utgangspunkt i brukertjenester, da implementasjonen av dette bruker de aller fleste av verktøyene og teknologiene. Grunnlaget for designet for autentisering og autorisering ble basert på en guide og tilpasset og sydd sammen basert på behov. Backend [93], Frontend [94]

4.6.1 Brukertjenester - Backend

Dette delkapittelet vil ta for seg en detaljert gjennomgang av implementasjonen av brukertjenester, fra et backend-perspektiv.

4.6.1.1 Registrering

POST-endepunkt som aksepterer et JSON-objekt. Dette JSON-objektet inneholder den ønskede informasjonen relatert til en bruker under registrering. Objektet blir behandlet som en parameter av typen signupRequest. SignupRequest er en modell klasse som ble laget for å behandle objektet som blir mottatt.

Når en registreringsforespørsel treffer endepunktet, skjer følgende prosess.
Ut ifra JSON-objektet tar den først ut brukernavnet og eposten, for å sjekke om det allerede eksisterer en bruker med disse brukerdetaljene, dersom det finnes en blir forespørselen ikke akseptert da disse må være unike.
Om forespørselen blir akseptert på dette steget går den videre og oppretter et nytt User-objekt basert på modell klassen dens og dens konstruktør. Informasjonen som blir gitt til denne konstruktøren for å lage objektet blir hentet ut fra JSON-objektet via den tidligere nevnte signupRequest modell klassen. Her blir også passord kryptert for å ikke lagre det i klartekst i databasen!

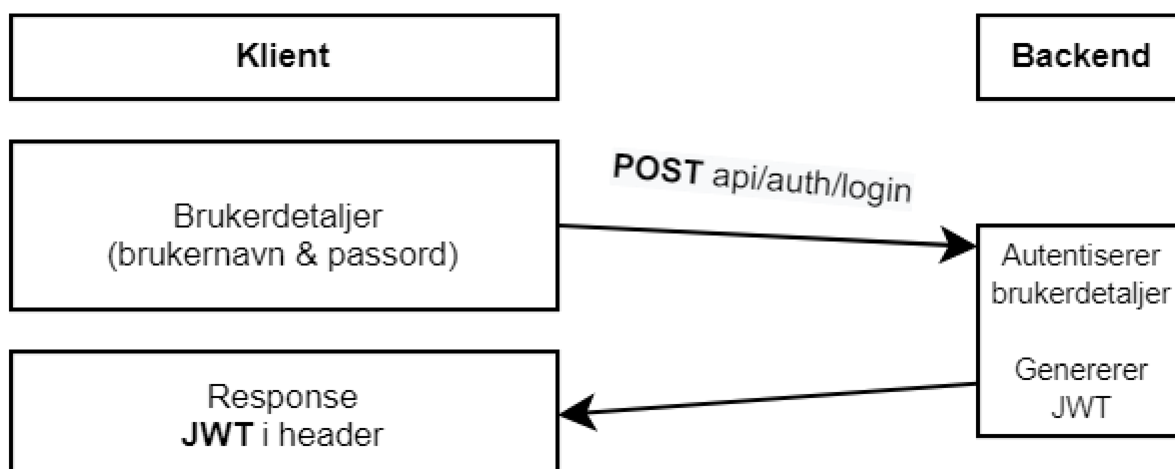
Gruppen valgte å benytte en gruppe/rolle basert autorisering, og dermed trenger brukerne en rolle. Rollene har en modell klasse og blir opprettet som Enum. Under

brukeropprettelsen vil den hente rollen ut ifra forespørselen, som blir av standard satt til å være User som er rollen for en vanlig bruker. Det opprettede brukerobjektet får deretter rollen satt til seg.

Etter rollen er blitt satt og alle ønskede brukerdetaljer er satt blir brukerobjektet sendt til userRepository, en repository klasse for brukertjenester for å bli lagret til databasen.
I likhet med de andre tjenestene som melding og turnering har brukertjenester en egen repository klasse. I Spring så kan man enkelt gjennomføre kall til databasen ved å benytte et Interface som implementerer JPA, en standard for implementering av håndtering av data mellom Java applikasjoner og databaser. I dette tilfellet under registrering er blir det ikke benyttet et eget, tilpasset kall da Spring Data JPA og JPA sin implementasjon har mange standard metoder den støtter.
Til slutt blir brukerobjektet som nevnt lagret til databasen.

4.6.1.2 Login & Autentisering

Informasjonsflyten for autorisering ved login er illustrert nedenfor i figur 53.



Figur 53 - Informasjonsflyt for autorisering ved login

POST-endepunkt som aksepterer kall som har et JSON-objekt i sin kropp. Dette JSON-objektet vil bli akseptert i kroppen basert på en modell klasse laget for å behandle innloggingsforespørsler, loginRequest, mye likt signupRequest fra registreringsprosessen.
Ut ifra dette objektet blir det hentet brukernavnet og passordet.
For å autentisere passordet benyttes **AuthenticationManager**, som er et Interface i Spring for å gjennomføre autentiseringer. I dette Interface-et benytter vi en klasse kalt **UsernamePasswordAuthenticationToken**, som gjennomfører selve autentiseringen, og returnerer et token basert på resultatet. Videre blir denne autentiseringen satt i et **SecurityContextHolder** objekt som gjør at konteksten av brukeren blir satt til autentisert slik at man kan gjennomføre videre sjekker av autentiseringen om mulig.

Når en bruker har blitt autentisert skal de også få generert et JWT token som kan bli brukt for autorisering og mer. Denne blir generert i en egen klasse kalt **jwtUtils**, detaljer om dette kommer i punkt nedenfor. Etter en bruker har autentisert seg får de i responsen litt informasjon. For laging av respons har gruppen benyttet egne klasser som bygger disse responsene. For innlogging blir det benyttet **jwtResponse**, som inneholder

en konstruktør som blir kalt i retur-delen av innloggingsmetoden. I denne responsen får brukeren returnert følgende:

- JWT Token
- Bruker ID, blir hentet via **userDetails.**
- Roller, blir hentet via **userDetails**.

4.6.1.3 Generering av JWT Token

Generering av et JWT token er ganske enkelt å bygge frem. Først må en definere to variabler.

- En secret - nøkkelord for generering av token
- Utløpstid - tid i millisekund.

I tokenet skal det bli bygd inn nødvendig brukerinformasjon, dette gjør at vi må få hentet frem denne informasjonen fra et sted. Dette gjør vi ved å benytte **userDetailsImpl**, som tillater oss å hente ut brukerdetaljer basert på autentiseringen. Videre blir det benyttet en egen builder som bygger selve tokenet basert på innstillingene vi velger. I gruppens tilfelle valgte vi å sette følgende innstillinger, som vist i figur 54.

```
return Jwts.builder()
        .setHeaderParam("typ", "JWT")
        .setSubject((authenticationPrincipal.getUsername()))
        .setIssuedAt(new Date())
        .setExpiration(new Date((new Date()).getTime() + jwtExpirationTime))
        .claim("uid", authenticationPrincipal.getUserId())
        .claim("username", authenticationPrincipal.getUsername())
        .claim("club", authenticationPrincipal.getClub())
        .claim("role", authenticationPrincipal.getAuthorities())
        .claim("email", authenticationPrincipal.getEmail())
        .signWith(SignatureAlgorithm.HS512, jwtSecret)
        .compact();
```

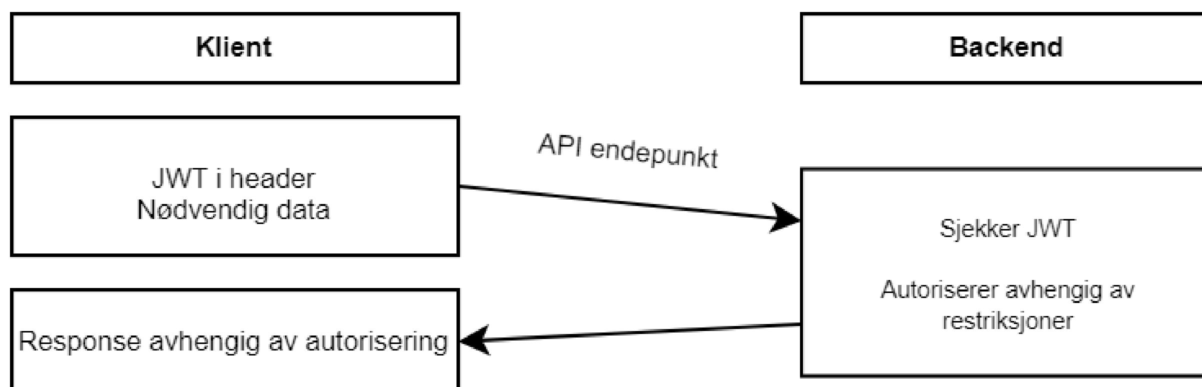
Figur 54 - Eksempel av bygging av JWT

Her blir det satt følgende innstillinger:

- setHeaderParam - Typen token.
- setSubject - setter emnet til tokenet, som blir satt til brukernavnet.
- setIssuedAt - når tokenet har blitt utstedt.
- setExpiration - når tokenet skal utløpe, dette blir satt til nåværende tidspunkt + utløpstiden som ble definert tidligere.
- .claim - dette er en måte man kan legge til informasjonen en ønsker å ha i tokenet. I gruppens tilfelle satte vi inn alt vi kommer til å trenge av informasjon på frontenden
    - Bruker ID, brukernavn, klubb, roller og epost.
- .signWith - setter hvilken algoritmer den skal bli kryptert med, samt hvilken secret den blir skrevet under med.

4.6.1.4 Autorisering for andre ressurser som krever autorisering

Figur 55 viser informasjonsflyten for autorisering for andre ressurser som krever autorisering.



Figur 55 - Informasjonsflyt for autorisering for andre ressurser som krever autorisering

APIen i gruppens backend er bygd opp i stor grad med autorisering på nødvendige ressurser. Ettersom vi har en del av ressursene som er tilgjengelige for alle er det en del som ikke er bak noen form for autorisering. Det er flere ressurser som er satt bak diverse autoriseringer, hvor du må ha en viss rolle for å benytte de.

Måten gruppen håndterer dette er ved at enhver bruker har en satt rolle til sin bruker. Under registrering vil denne bli satt som standard til «ROLE_USER». Denne rollen er en helt normal rolle hvor brukeren ikke har tilgang til noe annet enn en profiloversikt. Dersom det er en arrangør som skal ha en bruker kan en administrator endre dette via et eget Administratorpanel hvor de kan endre roller til enhver bruker så lenge de har bruker IDen, noe brukere finnes i profiloversikten sin.

Dersom en bruker er en arrangør og har fått endrer brukeren sin rolle til «ROLE_ORGANIZER». Vil brukeren nå ha tilgang til flere nye ressurser. Som nevnt i punkt om APIen sin struktur og hva en har tilgang til avhengig av ressurser vil brukeren nå kunne gjøre blant annet:

- Lage/Slette/Se sine turneringer
- Sende/Slette meldinger i de turneringene.

Dette er selvsagt ressurser og tjenester som ikke alle brukere skal kunne gjøre, og via den tidligere nevnte rollebaserte autoriseringen har gruppen fått gjennomført at kun brukere med nødvendig autorisering kan gjøre disse handlingene. Når en bruker logger inn vil det bli generert et JWT token som beskrevet i større detalj i punkt om login og autentisering. Dette tokenet inneholder nødvendig informasjon relatert til brukeren, blant annet rollen til brukeren. Slik at når en bruker forsøker å gjennomføre handlinger som trenger autorisering vil denne måtte bli sendt med. Når et kall når et endepunkt vil denne bli sjekket. Og basert på resultatet av sjekken til brukeren enten bli autorisert, eller ikke.

4.6.1.5 Administratortjenester

Når en bruker blir registrert så har blir den tilegnet rollen som en vanlig bruker, «ROLE_USER». For at brukere skal kunne opprette turneringer er de nødt til å ha rollen som arrangør, «ROLE_ORGANIZER». For å gjøre dette er det blitt opprettet en måte for Administratorbrukere å gjennomføre denne endringen. I **AuthController**, samme kontrolleren som håndterer innlogging og registrering er det opprettet et endepunkt for å gjennomføre denne handlingen.

```
@PutMapping("/updateRole")
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
public ResponseEntity<?> updateRole(@RequestParam int roleId, @RequestParam long userId) {
    var userDetails = (UserDetailsImpl) SecurityContextHolder.getContext().getAuthentication()
            .getPrincipal();
    Optional<User> userOptional = userRepository.findById(userId);
    if (userOptional.isPresent()) {
        var user = userOptional.get();
        if (userDetails.getUserId().equals(user.getUserId())) {
            return ResponseEntity.badRequest().build();
        }
    }
    userRepository.changeRole(roleId, userId);
    return ResponseEntity.ok(new MessageResponse("ok"));
}
```

Figur 56 - Implementasjon av metode for å endre en brukers rolle.

Følgende notasjoner er benyttet, som vist i figur 56:

- **@PutMapping("/updateRole")** - Definererer at endepunktet skal motta forespørsler av typen PUT, samt at stien den kan nås på er "*/updateRole", hvor "*" representerer URL-en i forkant.
- **@PreAuthorize("hasAuthority('ROLE_ADMIN')")** - Definerer at brukere som forsøker å nå endepunktet må ha autorisering som Administrator.

Videre ser vi at det kreves to parametere i metoden, som **@RequestParam.** Disse to parameterne er:

- roleId - IDen til rollen som brukeren sin rolle skal endres til.
    - Mer om hvordan rolle ID blir sendt forklares i Administratorpanel punktet under frontend.
- userId - IDen til brukeren som skal ha rollen endres.

Videre i metoden benytter vi **UserDetailsImpl** og **SecurityContextHolder** for å hente ut autentiseringen til brukeren. Deretter opprettes det en Optional for å sjekke om det eksisterer en bruker med den gitte IDen. Dette gjøres ved å benytte brukerens repository-klasse **userRepository,** og metoden *findById* som er en av de forhåndsdefinerte metodene. Denne ser etter brukeren og om den eksisterer blir det satt inn i den opprettede Optional at den eksisterer. Videre sjekker den om brukeren sin ID er samme som IDen til brukeren som sendte forespørselen, dersom den er det vil forespørselen bli nektet. Om den ikke er det blir det videre kalt en metode i **userRepository**, *changeRole()***.**

```
@Transactional
@Modifying
@Query(value = "update user_role set role_role_id = ?1 where user_user_id = ?2 ", nativeQuery = true)
void changeRole(int roleId, long userId);
```

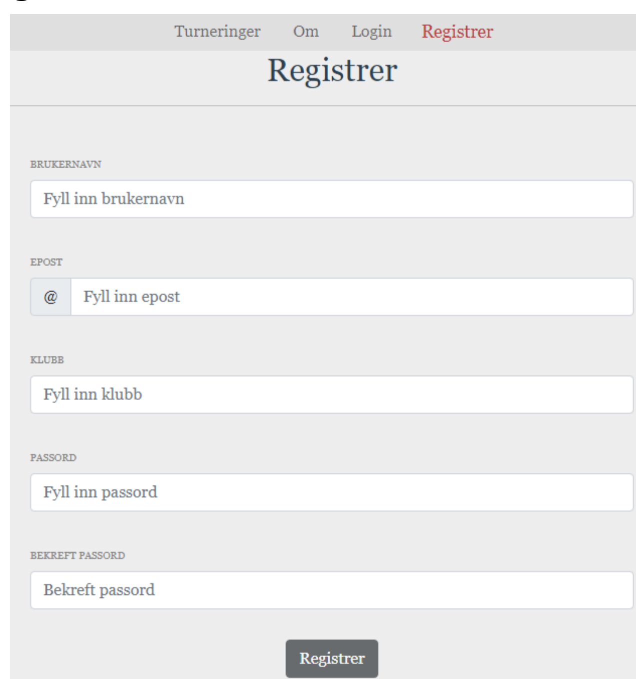
Figur 57 - Implementasjon av changeRole-metoden, med SQL-query

I denne metoden, vist i figur 57, blir det benyttet notasjonen **@Modifying**, denne må benyttes for å gjøre slik at **@Query** kan benytte andre handlinger enn kun SELECT, som er den eneste den kan gjøre uten denne notasjonen. Dette er nødvendig ettersom vi er nødt til å bruke en UPDATE handling. I metoden ser vi at det er to parameter, nemlig roleId og userId som nevnt ovenfor. I **@Query** notasjonen er selve handlingen som blir sendt til databasen. Her sier vi først hva vi vil gjøre - UPDATE, oppdatere tabellen "user_role". Deretter at vi skal sette role_role_Id til **?1** hvor user_user_id er **?2**. Parameterene kommer inn i bildet her. ?1 og ?2 representerer parameterne i metoden. ?1 er den første parameteren, her forteller vi at vi skal sette role_role_Id til hva det første parameteret, altså **roleId** er. ?2 er det andre parameteret, altså hvilken bruker som skal få endret sin rolle. Når dette er gjort og fremt brukeren og rollen eksisterer i databasen vil nå brukeren ha fått rollen sin endret.

4.6.2 Brukertjenester - Frontend

Dette delkapittelet vil ta for seg en detaljert gjennomgang av implementasjonen av brukertjenester, fra et frontend-perspektiv.

4.6.2.1 Registrering



Figur 58 - Registreringsskjema

Når en bruker ønsker å registrere seg kan de enkelt manøvrere seg til siden hvor de kan gjøre dette ved å trykke på "Registrer" knappen som er i navigasjonsbaren dersom brukeren ikke er autentisert. Når de trykker seg inn der blir de møtt av et skjema som

inneholder de nødvendige tekstboksene de på fylle inn for å registrere seg. Skjema er vist i figur 58.

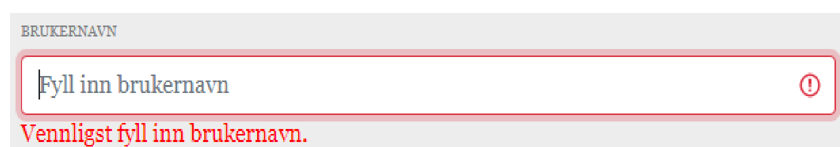
Validering

Hver av disse tekstboksene har diverse krav til lengde på backend-siden, dette er også gjort på frontenden, samt ekstra valideringer. Dette gjøres ved å benytte et rammeverk kalt **Vuelidate**. Det er viktig å validere dataen som blir skrevet inn i tekstboksene både for å sørge for at det er riktig type informasjon, men også for å sikre seg mot eventuelle problemer som kan oppstå f.eks. i databasen eller andre steder dersom det er symboler eller lignende som ikke er støttet i tekstboksene.
Valideringene som blir gjort på registrering er:

- Brukernavn
    - Kreves - Må fylles inn
    - Alfanumerisk - Kan kun ha bokstaver og tall.
    - Minimum og maksimum lengde, henholdsvis 3 og 25 tegn.
- Epost
    - Kreves - Må fylles inn
    - Epost - Må være formatert som en epost, altså inneholde "a@b.co" typ.
    - Maksimum lengde, 50 tegn.
- Klubb
    - Kreves - Må fylles inn
- Passord
    - Kreves - Må fylles inn
    - Minimum og maksimum lengde, henholdsvis 3 og 75 tegn
    - Sterkt passord
        - Må ha store og små bokstaver og et tall
-

Hver eneste av disse individuelle valideringene blir gjort av **Vuelidate** og dersom en av de returnerer med en error vil statusen til boksen, og dermed hele validering satt til ugyldig. I dette tilfellet vil send knappen bli deaktivert.

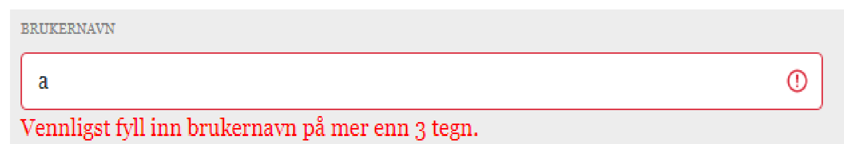
Ved hjelp av Vuelidate sin støtte for å returnere statusen til individuelle validering kan vi kombinere det med Bootstrap for å gi gode og tydelige tilbakemeldinger slik at de som prøver å registrere seg tydelig ser hva som eventuelt er feil. Dersom det er noe ugyldig med dataen i tekstboksen vil selve boksen bli markert i rødt og få et feilmeldings ikon i høyre hjørne, som vist i figur 59.



Figur 59 - Eksempel av felt som ikke har gyldig input, her inget brukernavn fylt inn

Vi kan også se at nedenfor boksen, i figur 59, står det i dette tilfellet "Vennligst fyll inn brukernavn". Ved hjelp av det tidligere nevnte feilmeldingsstatusene til Vuelidate kan vi endre denne basert på hvilken feilmelding det er. I bildet ovenfor klager den på at det mangler å fylles inn noe i boksen, men dersom det er fylt inn noe og den f.eks. er fort

kort i motsetning til ønsket lengde så vil den spesifisere at det er det feilen er. Den andre tilbakemeldingen er vist i figur 60.



Figur 60 - Eksempel av felt som ikke har gyldig input, for kort brukernavn

Videre registrering

Når en bruker har fylt inn skjemaet slik som ønsket uten noen feilmeldinger kan de sende inn skjemaet. Når brukeren trykker på knappen for å registrere seg kjører det funksjonen *sendUserData().* Funksjonen er vist i figur 61.

```
sendUserData() {
  var data = {
    username: this.v$.username.$model,
    email: this.v$.email.$model,
    club: this.v$.club.$model,
    password: this.v$.password.$model,
    role: ["user"]
  };
```

Figur 61 - Implementasjon av sendUserData-funksjon som henter ut validert data fra Vuelidate.

Her ser vi at først lager vi en variabel kalt **data**. Dette er hvordan vi bygger opp JSON-objektet som kreves på backenden for å registrere seg. I denne oppretter vi navnet på innholdet og henter disse inn ved hjelp av model-mapping som blir gjort i Vue.

```
this.v$.$validate();
if (!this.v$.$error) {
  alert("Registrering sendt");

  console.log(data);

  GameService.register(data)
    .then(response => {
      console.log("Resp: " + response);
      this.$router.push("/login");
    })
    .catch(error => {
      console.log("Err: " + error);
    });
} else {
  alert("Registrering kunne ikke bli sendt");
}
```

Figur 62 - Utsnitt av kode - Om data i kompomponent ikke inneholder feil, så kan man gjennomføre kall.

Deretter validerer den skjemaet og dersom den finner noe feil sender den et varsel til brukerens nettleser og returnerer uten å sende forespørselen. Dette er vist i figur 62. Dersom den ikke finner noe feil får brukeren et varsel om at skjemaet ble sendt. Deretter blir det kalt en funksjon fra **GameService**, et lite dekkende nav, som bygger opp selve kallet som blir kjørt til backenden.

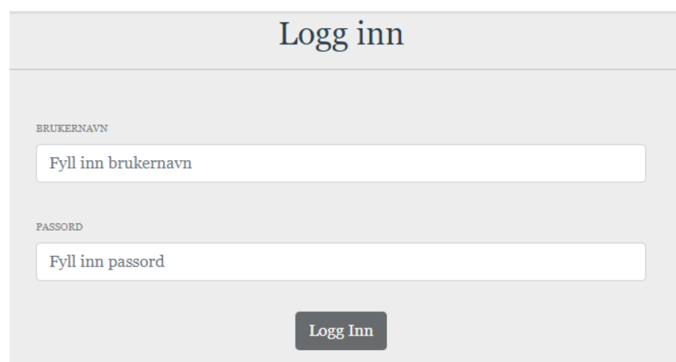
```
register(data) {
  return apiClient.post(registerURL, data);
}
```

Figur 63 - Implementasjon av register-funksjonen i service-klassen GameService.

Denne funksjonen, vist i figur 63, er i seg selv veldig enkel da den tar instansen av Axios og bygger videre på den. Her ser vi at selve funksjonen tar en parameter, **data**. Dette parameteret er JSON-objektet vi lagde ovenfor som inneholder brukerdetaljene. Videre i funksjonen blir det returnert den tidligere nevnte Axios instansen, men nå som en POST forespørsel med to parametere. Disse to parameterne er først URL-en til hvor forespørselen skal bli sendt, og siste er det nevnte JSON-objektet.

Dersom det blir godtatt på backenden og man får en respons vil brukeren deretter bli sendt videre til login for innlogging.

4.6.2.2 Innlogging



Figur 64 - Innloggingsskjema

Når en bruker enten har registrert seg eller ønsker å logge seg inn kan de gå til innloggingsskjemaet, vist i figur 64, for å gjøre dette. Her blir de møtt av to tekstbokser de er nødt til å fylle inn.

Validering

Slik som beskrevet mer i detalj i punkt om Registrering blir tekstboksene validert slik at dataen er fylt inn som ønsket. Valideringene på login er mindre strenge enn på registrering som følge av at det er innsending av eksisterende brukerdetaljer i form av brukernavnet og passordet som skal bli autentisert.
Valideringene som blir gjennomført er

- Brukernavn & Passord
    - Kreves - må fylles inn

I likhet med registrering vil tekstboksene sitt utseende og feilmeldingen under de endre seg avhengig av feilmeldingen. Samt vil "logge inn" knappen bli deaktivert dersom det er noe ugyldig med skjemaet.
Dersom alt er gyldig, kan bruker trykke på login for å sende forespørselen.

Sending av forespørsel

Sendingen av innloggingsforespørselen er litt mer komplisert for innlogging enn for registrering. Dette er av et par årsaker. Først blir Vuex benyttet for å lagre et JWT token som blir sendt i respons ved suksessfull innlogging. Og i motsetning til registrering blir ikke **GameService** benyttet i dette tilfellet av samme årsak. Istedenfor blir selve forespørselen laget som en **action** i Vuex.

```
methods: {
  ...mapActions("auth", {
    actionLogin: "login"
  }),
```

Figur 65 - Eksempel av bruk av mapActions

Vi kan se av i **methods**, i figur 65, er det en funksjon deklarert, **"...mapActions()"** som er måten en henter ut og mapper definerte **actions** i Vuex for å benytte de andre steder i koden.

```
async login({ commit }, payload) {
  const response = await GameService.login(payload).catch(err => {
    console.log(err);
  });
  if (response && response.data) {
    commit("saveTokenData", response.data);
    commit("setLoginStatu", "success");
  } else {
    console.log("failed");
    commit("setLoginStatu", "failed");
  }
},
```

Figur 66 - Implementasjon av asynkron login-funksjon i Vuex modul.

Ovenfor, i figur 66, er innloggingsfunksjonen, som er asynkron. I denne er det et par parameter, den første er **commit** som er måten en spesifiserer at det skal bli gjort en endring, eller **mutation** av noe i Vuex. **Payload** er innholdet som skal bli sendt i forespørselen, som i dette tilfellet vil være brukernavnet og passordet til brukeren som skal logge inn. Videre blir det kalt login funksjonen fra **GameService** for å sende selve forespørselen. Videre handling varierer avhengig av responsen. Dersom innloggingen ikke var suksessfullt, f.eks. feil passord e.l., vil det bli returnert en error og loginstatus blir satt til failed. Dersom responsen var OK og brukeren har fått logge inn blir det gjennomført to handlinger, første er lagring av JWT token, som blir forklart mer i neste punkt, samt login status blir satt til success.

```
async login() {
  await this.actionLogin({
    username: this.v$.username.$model,
    password: this.v$.password.$model
  });
  if (this.getterLoginStatus == "success") {
    this.$router.push("/profil");
  } else {
    alert("failed to login");
  }
```

Figur 67 - Implementasjon av asynkron login-funksjon i Login-komponent

Ovenfor er selve funksjonen som blir kalt når en bruker forsøker å logge inn, i denne blir den funksjonen nevnt ovenfor fra Vuex kalt med linjen "*await this.actionLogin()*". Her blir brukernavnet og passordet sendt videre til den funksjonen for å sende forespørselen. Og avhengig av responsen blir brukeren enten varslet om at den ikke fikk logget inn, eller så dersom forespørselen blir godkjent vil de være logget inn og bli sendt videre til profilen deres.

4.6.2.3 Lagring av JWT med Vuex

JWT er nødvendig for å kunne nå ressurser som krever autentisering, og dette må på en måte bli lagret et sted. Dette løste gruppen ved å benytte Vuex. Under punktet om sending av forespørsel ovenfor er det en linje som inneholder følgende kode, vist i figur 68:

```
commit("saveTokenData", response.data);
```

Figur 68 - Eksempel av hvordan man lagrer til Vuex

Denne linjen med kode er essensiell for å lagre tokenet. Som nevnt blir **commit** benyttet for å endre, eller **mutere** en tilstand eller data i Vuex. I denne linjen er «saveTokenData", denne linjen representerer en **mutation** som er laget i Vuex filen **auth.js**.

```
const mutations = {
  saveTokenData(state, data) {
    localStorage.setItem("access_token", data.access_token);
    localStorage.setItem("refresh_token", data.refresh_token);

    const jwtDecodedValue = jwtDecrypt(data.access_token);
    const newTokenData = {
      token: data.access_token,
      refreshToken: data.refresh_token,
      tokenExp: jwtDecodedValue.exp,
      userid: jwtDecodedValue.sub,
      uid: jwtDecodedValue.uid,
      role: jwtDecodedValue.role[0].authority,
      club: jwtDecodedValue.club
    };
    state.authData = newTokenData;
```

Figur 69 - Implementasjon av saveTokenData som lagrer JWT og informasjon det inneholder.

I denne funksjonen, vist i figur 69, blir det gjort en del. Først får en sendt med to parameter, sin state og dataen, data er responsen fra backenden som inneholder tokenet. Videre ser vi måten gruppen har valgt å lagre JWT tokenet, som er ved å benytte en nettleser sitt lokale lagringssystem, **localStorage.** Via .setItem kan vi enkelt lagre ønsket informasjon her, spesifisere navnet på lagret data og hva slags data, som vi i dette tilfellet henter ut ifra responsen.

Videre blir JWT tokenet dekryptert ved hjelp av en typisk dekrypteringsklasse. Og deretter blir det opprettet et objekt som tar inn og lagrer denne dekrypterte dataen. Som deretter blir lagret til **authData**, som er et objekt i Vuex klassen for å lagre staten til noe.

Når dette er gjort er da JWT tokenet lagret for å kunne bli benyttet videre, samt er det enkelt å kunne gjøre rollebasert rendering av objekter da man kan hente ut innholdet fra **authData** ved å importere Vuex inn i ønsket klasse. Dette blir diskutert mer i punkt lenger ned

4.6.2.4 Autorisering via JWT/Vuex

Vuex tillater en å enkelt kunne administrere og ha tilgjengelig nødvendig data. Og som nevnt i punktet ovenfor valgte gruppen å benytte Vuex for å lagre JWT token for videre autorisering. Ettersom gruppen valgte å benytte et autoriseringssystem som skiller på tilgjengelige ressurser avhengig av hvilken rolle en bruker har måtte vi ha en måte å få lagret dette på. Måten en kan få hentet ut denne informasjonen er ved å benytte **getters** som er en av denne Vuex lageret. I selve Vuex filen **auth.js** er alle disse «getterne» definert, og er de som skal bli kalt dersom en ønsker å hente informasjon relatert til noe lagret med Vuex. Gruppen har brukt dette flere steder, i stor grad på de rollebaserte tjenestene som vil bli gjennomgått i dypere detalj i neste punkt.

4.6.2.5 Dynamisk innlasting

Etter en bruker har autentisert seg vil de få i respons et JWT token som blir lagret i nettleserens lokale lager. Som beskrevet i punkt blir det hentet ut mye informasjon fra denne, deriblant rollen til en bruker, noe som er essensielt for prosjektet ettersom mye er basert på hvilken rolle en bruker har. En ikke-innlogget og en innlogget bruker trenger ikke å ha samme tilgjengelige de samme ressursene, f.eks. så trenger ikke en innlogget bruker å ha login eller registreringsknapp i navigasjonsbaren. Dette løste vi ved å benytte **getters** i Vuex. Somnevnt i punkt over gir er getters måten man kan hente ut informasjon fra noe lagret med Vuex, og for å hente ut innloggingsstatusen til en bruker benyttet vi dette.

```
isTokenActive(state) {
  if (!state.authData.tokenExp) {
    return false;
  }
  return tokenAlive(state.authData.tokenExp);
},
```

Figur 70 - Implementasjon av funksjonen isTokenActive

Funksjonen ovenfor *isTokenActive(),* vist i figur 70*,* er den vi benyttet for å sjekke om noen er innlogget. Måten dette funker er ganske rett frem, den kjører i et if-statement en sjekk av et innhold av **authData**, tokenExp er falsk eller sann. Og avhengig av denne statusen returnerer den falsk eller sann.

Per nå er denne getter funksjonen kun i den lokale **auth.js** filen, og for å kunne benytte denne andre steder i prosjektet er vi nødt til å importere inn **store** og **mapGetters.** **Store** er begrepet Vuex bruker for å beskrive sitt eget lager, hvor alt man definerer blir lagret. **mapGetters** er en måte vi kan hente ut funksjonene fra getters i dette lageret og tilegne de et visst variabel som vi kan kalle i koden. Man kan også spesifikt hente ut en variabel inne i en funksjon, noe vi også benyttet.

```
computed: {
  getLoginStatus() {
    return store.getters["auth/isTokenActive"];
  },
  ...mapGetters("auth", {
    gettersAuthData: "getAuthData"
  })
},
```

Figur 71 - Implementasjon av getLoginStatus-funksjon og mapping av funksjoner fra Vuex store til komponent

Ovenfor, i figur 71, ser en begge disse måtene, for å sjekke statusen for innlogging benyttet vi funksjonen *getLoginStatus()***,** som gjør et kall til getteren *isTokenActive* som en kan se i bildet lenger opp. Denne funksjonen kan vi nå benytte ellers i koden for å kunne dynamisk laste inn ting avhengig av statusen. Vue.js tillater å gjøre dette på en veldig enkel måte ved å bruke noe kalt «v-if».

```
<li class="nav-item">
  <router-link v-if="!getLoginStatus" to="/login" class="nav-link"
    >Login</router-link
  >
</li>
```

Figur 72 - Eksempel av dynamisk innlasting av Router-link.

Figur 72, viser måten vi har gjennomført dynamisk innlastning. I denne linjen har vi plasser inn *«v-if="!getLoginStatus"»*. Dette gjennomfører er sjekk av funksjonen forklart ovenfor og dersom den returnerer *false*, altså at brukeren ikke er innlogget vil den laste inn denne lenken i navigasjonsbaren, og om brukeren er innlogget blir den ikke lastet inn. Lignende er gjort med registrering og profil.

4.6.2.6 Ruting med autentisering

I vår løsning bruker vi Vue Router for å rute mellom komponentene i siden. I en egen klasse blir alle de forskjellige rutene definert med navn sti, navn og hvilken komponent det gjelder. Man kan også legge til metadata her, i vårt tilfelle er det lagt til «requiredAuth» og avhengig av nødvendigheten for autentisering er denne satt til sann eller falsk. En definisjon av en rute som krever autorisering er vist i figur 73.

```
{
  path: "/admin/panel",
  component: AdminPanel,
  meta: {
    requiredAuth: true
  }
```

Figur 73 - Eksempel av definisjon av rute hvor det kreves autorisering

For et ekstra trinn med sikkerhet, og for å unngå at folk går til steder de ikke skal kunne, eller ikke trenger å kunne gå ved hjelp av å spesifikt gå til en URL kan man benytte diverse måter å løse dette på. Vue-rutere har en funksjon kalt *beforeEach* som gjennomføres når en ruting blir gjort.

```
router.beforeEach((to, from, next) => {
  if (!store.getters["auth/getAuthData"].token) {
    const access_token = localStorage.getItem("access_token");
    const refresh_token = localStorage.getItem("refresh_token");
    if (access_token) {
      const data = {
        access_token: access_token,
        refresh_token: refresh_token
      };
      store.commit("auth/saveTokenData", data);
    }
  }
```

Figur 74 - Eksempel av bruk av beforeEach-funksjon for ruting, del 1

I denne funksjonen, vist i figur 74, er det første som blir gjort er en sjekk av om det eksisterer et JWT token i nettleserens lokale lagring, som skal være der dersom brukeren er innlogget, og autentisert.

```
const auth = store.getters["auth/isTokenActive"];

if (to.fullPath == "/") {
  return next();
} else if (to.fullPath == "/login" && auth) {
  return next({ path: "/profil" });
} else if (to.fullPath == "/register" && auth) {
  return next({ path: "/profil" });
} else if (!auth && to.meta.requiredAuth) {
  localStorage.clear();
  return next({ path: "/login" });
}

return next();
```

Figur 75 - Eksempel av bruk av beforeEach-funksjon for ruting, del 2

Videre ser man i figur 75 at det blir opprettet en konstant **auth** som bruker en funksjon i Vuex lageret som returnerer sann eller falsk statusen på om et token er aktivt. Nedenfor dette ser vi en del forskjellige if-spørringer. De to øvre "else-if" spørringene gjennomføres dersom stien til der brukeren forsøker å gå er /login eller /register. Her blir det sjekket om brukeren er autentisert. Dersom de er det vil brukeren bli sendt til /profil

istedenfor, ettersom en innlogget bruker ikke trenger å logge inn, eller registrere seg. Nederst er det en funksjon som sjekker om brukeren ikke lenger er autentisert, og om de forsøker å gå til en komponent som krever dette. Her blir det da slettet eventuelle token som ligger igjen i localStorage, og brukeren blir sendt til /login for å autentisere seg på nytt.

4.6.2.7 Autorisering med roller

Profil - lasting av adminknapp.

For å gjennomføre autorisering har vi benyttet et lignende system med Vuex og getters. Når en bruker har rollen som administrator vil den i sin profil kunne trykke seg videre inn i et Administratorpanel, dette panelet vil bli mer beskrevet i et senere punkt. Måten dette er løst på er ved å hente ut data fra Vue lageret.

```
const getters = {
  getAuthData(state) {
    return state.authData;
  },
```

Figur 76 - Eksempel av implementasjon av getters-funksjon

```
computed: {
  ...mapGetters("auth", {
    gettersAuthData: "getAuthData"
  }),
```

Figur 77 - Eksempel av mapping av funksjoner fra Vuex

I figur 76 ser man funksjonen fra **auth.js** som returnerer **authData**, som er et objekt som inneholder informasjonen hentet ut fra JWT tokenet. I figur 77 er måten denne blir hentet inn og tilegnet under et definert variabelnavn **gettersAuthData** inne i **Profil.vue** filen. Her tar vi nå igjen og benytter «v-if:» for å gjøre dynamisk innlasting.

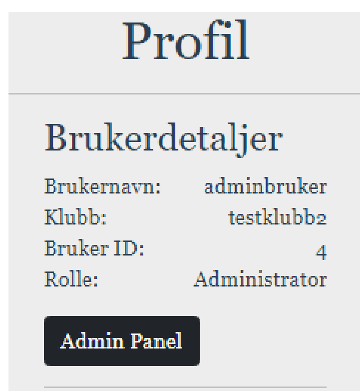
```
<div v-if="gettersAuthData.role === 'ROLE_ADMIN'" class="admin-button">
  <button
    v-if="gettersAuthData.role === 'ROLE_ADMIN'"
    class="btn btn-dark"
    @click="$router.push('/admin/panel')"
  >
    Admin Panel
  </button>
</div>
```

Figur 78 - Eksempel av dynamisk innlasting av elementer

Ovenfor, i figur 78, så ser vi lignende «v-if:» som tidligere, men i dette tilfellet blir det ikke kjørt en funksjon i JavaScript delen av filen, her blir det i stedet gjennomført en sjekk i selve «v-if:» utsagnet. Her tar den da og henter informasjonen i objektet den henter fra Vuex lageret og sjekker om **role** elementet i dette objektet stemmer overens med kravet, «**ROLE_ADMIN**». Og avhengig av dette vil den laste inn seksjonen som inneholder en knapp som videre linker en videre til Administratorpanelet. Dersom en

bruker er autorisert som Administrator vil den ved trykk på knappen bli sendt videre til Administratorpanelet via $router.push, som er måten man kan sende en til et annet sted i nettsiden og laste inn et nytt element i Vue. I figur 79 vil se hvordan profil ser ut om du er vanlig bruker og i figur 80 hvordan den ser ut om du er Administrator.

Figur 79 - Visning av profil med for andre brukere



Figur 80 - Visning av profil for Administrator

Henting av eide turnering for arrangør

For en arrangør er det også ordnet slik at de ser på profilen sin, hvilke turneringer de selv er eiere og arrangører av. Dette er gjort på lignende måte ved hjelp av Vuex getters og «v-if» sjekker, samt kall mot backenden. Når en bruker går inn på profilen sin og fremt brukeren er en arrangør vil det bli gjort et kall mot en funksjon i komponenten.

```
methods: {
  getTournamentsByOwner() {
    GameService.getTournamentsByOwner(this.gettersAuthData.uid)
      .then(response => {
        this.tournaments = response.data;
      })
      .catch(error => {
        console.log(error);
      });
  },
```

Figur 81 - Implementasjon av getTournamentsByOwner-funksjon.

Denne funksjonen, vist i figur 81, gjennomfører et kall som er en del av **GameService** klassen. Her henter den ut bruker IDen fra **gettersAuthData**, tilsvarende måte som gjort ovenfor med innlastning av Administratorknapp. Og sender det som en parameter

til backenden som returnerer turneringer som er eid av den IDen, om noen. Her blir da en variabel “tournaments”, som er forhåndsdefinert som et tomt array, satt til dataen som er i responsen.

```
<div v-if="hasTournaments" class="profile-content">
  <div
    v-if="
      gettersAuthData.role === 'ROLE_ORGANIZER' ||
        gettersAuthData.role === 'ROLE_ADMIN'
    "
  >
    <TournamentList :tournaments="tournaments" title="Mine Turneringer" />
  </div>
</div>
```

Figur 82 - Eksempel av dynamisk innlastning elementer basert på bruker har turneringer og er enten Administrator eller arrangør

Dersom brukeren har turneringer den er eier/arrangøren av vil dette gjøre av det blir lastet inn en div boks. Deretter blir det gjennomført en lignende sjekk som på knappen for Administratorpanel hvor den sjekker rollen til brukeren og dersom den har riktig rolle vil den laste inn turneringene til brukeren. Dette er vist i figur 82.

Profilen til denne brukeren vil da inneholde en liste av brukerens turneringer, samt muligheten til å gå til et eget Dashboard for turneringene. Dette er vist i figur 83.

Figur 83 - Eksempel på visning av egne turneringer.

4.6.2.8 Header i forespørsel for autorisering

Tidligere har det blitt benyttet en egen sentralisert klasse, **GameService** for å sende forespørsler til backenden. Denne fungerer fint til forespørsler hvor det ikke kreves noen form for autorisering. Ettersom det finnes en del forespørsler som krever autorisering måtte det implementeres en ny måte å sende disse på. Av denne grunn ble klassen **AuthoService** opprettet. I denne klassen blir det først opprettet en instans av Axios klienten, her vist i figur 84.

```
const instance = axios.create({
  baseURL: baseURL_API
});
```

Figur 84 - Eksempel på oppretting av Axios instans

Videre blir det benyttet "**interceptors"**. Interceptors tillater enn å kunne kjøre en bulk med kode FØR forespørselen blir sendt. Dette er essensielt da vi må få hentet ut JWT tokenet fra localStorage for å få det sendt med i headeren for å få kunne gjennomført autorisering.

```
instance.interceptors.request.use(
  config => {
    const token = localStorage.getItem("access_token");
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  error => Promise.reject(error)
);
```

Figur 85 - Konfigurasjon av Axios instans med interceptor

I figur 85 ser man at det blir opprettet en konfigurasjon for interceptoren skal bruke ved forespørsler. I denne blir det opprettet en konstant kalt token, her blir JWT tokenet som er lagret i localStorage bak navnet "access_token" hentet ut og lagt inn i den. Fremt den eksisterer blir den da satt til å være en header, og en autoriserings header. Ettersom det er Bearer token blir dette spesifisert her også. Videre blir dette returnert, og vi har nå en instans av Axios klienten som kan bli brukt for å sende forespørsler med autorisering. Videre i denne klassen er det definert funksjoner for de forskjellige typene kall som skal bli sendt, GET, POST osv. Disse har blitt laget etter nødvendigheten for hvordan de blir opprettet og sendt ellers i koden. Eksempel av funksjonen for GET-forespørsler er vist nedenfor i figur 86.

```
get(url) {
  return instance
    .get(url)
    .then(response => response)
    .catch(reason => Promise.reject(reason));
},
```

Figur 86 - Implementasjon av generell get-funksjone for autorisering.

GET, PUT og DELETE er laget på tilsvarende måte med at alle tre har kun en enkelt parameter som blir sendt til den, som er URL-en. Årsaken til dette på PUT og DELETE er at det er RequestParam som blir benyttet for de forespørslene, og det blir en den av URL-en. Videre inn i funksjonen blir det definert av det skal bli gjort et kall av typen GET, her blir det definert PUT i PUT funksjonen, og DELETE i DELETE funksjonen. Etter dette inneholder de to neste funksjonene hva som skal bli gjort med svar, hvor respons blir sendt inn i respons for å kunne håndtere den. POST er rimelig lik som de andre funksjonene, men her er det lagt til en andre parameter, som representerer objektet med data som skal bli sendt med.

4.6.2.9 Opprettelse av turnering

Figur 87 - Visning av opprettning av turnering.

Når en bruker er autorisert som en arrangør ved hjelp av en sjekk av rollen deres hentet fra **authData**. Vil brukeren få tilgjengelig en knapp på hjemmesiden, vist i figur 87, de kan trykke på for å gå videre til et skjema hvor de kan opprette turneringer. Her blir de møtt av følgende skjema, vist i figur 88.

Opprett en turnering

Navngi din turnering og arbiter

TITTEL

Tittel...

NAVN PÅ ARBITER

Navn...

Bestem start- og sluttdato for turneringen

STARTDATO

2021-05-14

SLUTTDATO

Sluttdato (Klikk for å velge)

Tilbake Reset

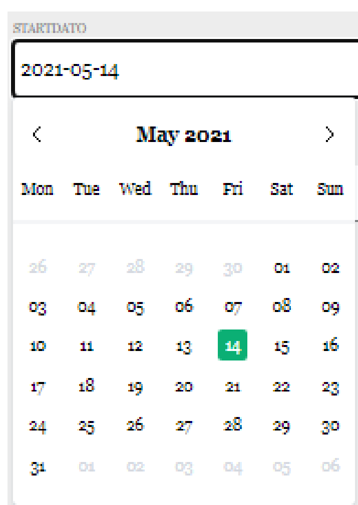
Opprett

Figur 88 - Skjema for oppretting av turnering

Når en turnering skal bli opprettet kreves det spesifikke informasjoner som skal måtte bli fylt ut. Disse er:

- Tittel - Tittelen på turneringen
- Arbiter - Navn på arbiter
- Startdato - Når turneringen starter
- Sluttdato - Når turneringen avsluttes.

Tittel og arbiter feltene er tradisjonelle tekstfelt, imens for feltene for dato er det brukt en ferdiglaget komponent kalt "DatePicker", vist i figur 89, som gir muligheten for å velge dato fra en kalender istedenfor å måtte skrive dette inn manuelt.



Figur 89 - Visning av en dato-velger.

Validering

Validering er gjort på alle disse tekstfeltene. Følgende validering blir gjort:

- Tittel
  - Kreves - må fylles inn
- Arbiter
  - Kreves - må fylles inn
  - Alfanumerisk - må være alfanumerisk
- Startdato
  - Kreves - må fylles inn
- Sluttdato
  - Kreves - må fylles inn

Videre sending

Når all informasjon er fylt inn som ønsket kan brukeren trykke på knappen for å sende informasjonen og opprette en turnering. Ved trykk på knappen blir funksjonen *sendForm* kallet.

```
var tournament = {
  tournamentName: this.v$.tournamentName.$model,
  startDate: this.v$.startDate.$model,
  endDate: this.v$.endDate.$model,
  arbiter: this.v$.arbiter.$model
};
```

Figur 90 - Kode som lagrer validering input for en turnering.

Det første som blir gjort her er at det blir opprettet et objekt som henter ut informasjonen fra tekstboksene og setter de til å tilhøre et element i objektet, vist i figur 90. Dette er JSON-objektet som skal bli sendt i kroppen til forespørselen.

```
AuthoService.post("tournaments/createtournament", tournament)
  .then(response => {
    if (response.data) {
      const path = "/dashboard/" + response.data.id;
      this.routingToDashboard(path);
    } else {
      this.error = true;
    }
  })
  .catch(err => {
    console.log(err);
    this.error = true;
  });
this.resetForms();
```

Figur 91 - Implementasjon av post-funksjon for opprettelse turnering

Videre blir det benyttet **AutoService** klassen da det kreves autorisering for å kunne legge til en turnering. Her blir *post* funksjonen i klassen brukt da det er en POST forespørsel som kreves. I denne funksjonen kreves det to parameter, URL - hvor skal den sendes, og det andre parameteret er dataen som skal bli sendt, som er JSON-objektet opprettet ovenfor. Her blir forespørselen da sendt til backenden. Dersom den går gjennom OK og blir lagt inn vil brukeren kunne bli sendt direkte til turneringen sitt Dashboard. Dette gjøres ved å hente ut dataene fra responsen, i denne responsen er IDen til turneringen vedlagt og kan enkelt hentes ut. Ettersom URL-en til en turnering sitt Dashboard er "/dashboard/{id}" kan vi enkelt bygge denne URL-en ved hjelp av den returnerte IDen fra responsen. Denne URL-en blir da sendt til en funksjone som inneholder en router.push funksjone som sender brukeren til dashbordet. Implementasjonen er vist i figur 91.

4.6.2.10 Turnerings-Dashboard

Dashboard
The Easter Chess Championship 2021
Sendte meldinger
Velkommen til Per Atle Steinars Toten Turnering
Slett
test4
Slett
MELDING
Fyll inn melding
VIKTIGHET
Velg viktighet...
Send
Slett turnering
Tilbake

Figur 92 - Eksempel av Dashboard for en turnering

En arrangør vil kunne gå inn i oversikten over sine turneringen, og der kan de gå videre til et dashboard hvor de kan få gjøre diverse tjenester relatert til den gitte turneringen. Dette dashboardet er illustert i figur 92. For å legge til et ekstra steg med sikkerhet har vi benyttet flere sjekker i innlastingen av denne siden for å begrense hva en man gjøre

dersom man går rett til lenken for en turnering sin dashboard, uten å være eier eller arrangør. Dette gjøres på to steder på hoved div-boksen for siden, i denne blir det gjennomført en «v-if:» sjekk, vist i figur 93, med to betingelser i form av funksjoner, vist i figur 94 og figur 95.

```
<div v-if="adminOrOwner && ownerOfTournament">
```

Figur 93 - Eksempel av betinget visning av element

```
ownerOfTournament() {
  if (this.tournament != "") {
    return (
      this.$store.state.auth.authData.uid == this.tournament.owner.userId
    );
  } else {
    return false;
  }
},
```

Figur 94 - Implementasjon av ownerOfTournament-funksjon

```
adminOrOwner() {
  return (
    this.gettersAuthData.role === "ROLE_ORGANIZER" ||
    this.gettersAuthData.role === "ROLE_ADMIN"
  );
},
```

Figur 95 - Implementasjon av adminiOrOwner-funksjon.

Først blir det sjekket rollen til brukeren, her om brukeren har nødvendig autorisering, som i dette tilfelle vil være enten arrangør, eller administrator. Samtidig med dette blir det gjennomført en sjekk av turneringen den forsøker å gå inn på dashbordet til. Her sjekker den først om turneringen eksisterer. Dersom den eksisterer sammenligner den bruker IDen til den innloggede brukeren med bruker IDen som tilhører turneringen. Dersom begge disse to betingelsene blir evaluert til sann vil dashbordet bli tegnet. Om ikke, vil de bli møtt av et vindu om at de ikke er autoriserte eller ikke er eieren av turneringen. Et eksempel av dette ser man i figur 96.

Figur 96 - Eksempel på visning av beskjed gitt i Dashboard til bruker dersom om brukeren ikke er eier av turneringen

Se meldinger

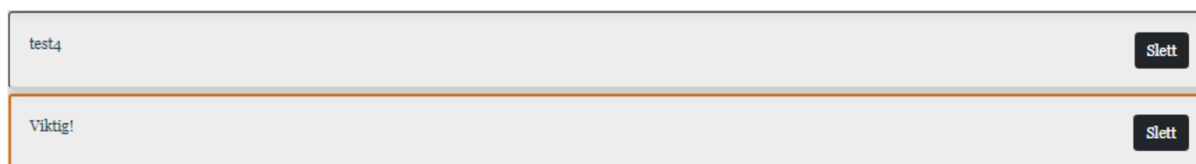
Videre inn i selve dashbordet har de et par tilgjengelige tjenester. Som beskrevet i mer detalj i et punkt lenger opp har hver eneste turnering meldinger som kan bli sendt av arrangøren og sett av tilskuerne. I Vue så kan man benytte **views** og **components** som representerer sider og komponenter man kan koble sammen og sette inn i en side. På denne måten blir meldingsdelen lastet inn i dashbordet. Visningen av meldinger, og sending av meldinger er hver sin komponent som blir lastet inn. For å hente inn meldinger benyttes komponenten **DashboardMessage.vue**, inne i denne gjennomføres

det et kall mot backenden som henter alle meldingene for turneringen. Funksjon som gjør dette kallet er vist i figur 97.

```
fetchMessages() {
  GameService.getMessagesForTournament(this.tournamentid)
    .then(response => {
      this.messages = response.data;
    })
    .catch(error => {
      console.log(error);
    });
```

Figur 97 - Implementasjon av fetchMessage-funksjon

Når meldingene har blitt mottatt i responsen vil vi sende denne dataen videre til en egen underkomponent som har ansvaret for å laste inn hver enkelt av meldingene. Dette gjøres ved å benytte **«props»**, et argument som kan gis en komponent, som vi kan benytte for å spesifisere data som skal kunne bli transportert mellom komponenter. Og ved hjelp av eksportering og importering av det kan vi nå sende nødvendig data til korrekt plass. **MessageView.vue** er komponenten som har ansvaret for å gjennomføre den nevnte innlastingen av disse meldingene. Her tar den da og sjekker gjennom meldingen, som kommer i formen av et array. Denne dataen blir da satt til å bli vist på korrekt plass i en liste.



Figur 98 - Visning av sendte meldinger i Dashboard.

I figur 98 ser vi at innholdet til meldingene har blitt lastet inn, med innholdet plassert i kortet. Vi kan se at det forskjell på disse to meldingene sitt utseende, da ene har en oransje kantlinje rundt seg, dette er for å markere viktigheten til meldingen. Ettersom noen meldinger vil være viktigere enn andre har vi implementert en måte dette kan bli markert. Dette gjøres ved at i dataen til meldingen er det markert viktigheten dens. Når en melding blir lastet inn gjøres det ved at den laster inn meldingene med kode vist i figur 99.

```
<div
  class="message-card"
  :class="{ important: important, overflowable: overflowable }"
>
```

Figur 99 - Eksempel på elementet som viser meldingene.

Her ser en at det er definert flere klasser for elementet. Elementet får klassene important og overflowable om verdiene av tilhørende data i komponenten blir evaluert til sann. Viktigheten til meldingen blir sjekket av en egen funksjon som blir kjørt når en melding blir lastet inn. Funksjonen, vist i figur 100, ser om viktigheten av meldingen som er satt i objektet er «viktig», og dersom den er det blir important satt til true, og meldingen blir lastet inn som viktig og får den oransje kantlinjen.

```
setImportance() {
  if (this.message.importance.toString().toLowerCase() == "viktig") {
    this.important = true;
  }
```

Figur 100 - Implementasjon av setImportance som bestemmer om en melding er viktig eller ikke i frontend.

Slette meldinger

Vi kunne i figur 98 se at det er en knapp til høyre for meldingene hvor det står «slett». Dette er en knapp som tillater en å kunne slette en melding, dersom dette skulle være ønskelig å gjøre.

```
<div v-if="deleteButton">
  <button
    v-if="
      gettersAuthData.role === 'ROLE_ORGANIZER' ||
        gettersAuthData.role === 'ROLE_ADMIN'
    "
    class="btn btn-dark delete-message-button"
    @click="deleteMsg"
  >
    Slett
  </button>
</div>
```

Figur 101 - Implementasjon av slette-knapp for meldinger

Denne knappen blir dynamisk lastet inn avhengig av rollen til brukeren, dette vist i figur 101. Ved riktig rolle vil knappen kunne bli lastet inn og brukeren vil kunne sende en forespørsel om å slette meldingen. Når knappen blir trykket på blir det gjort et kall mot en funksjon definert i komponenten, vist i figur 102.

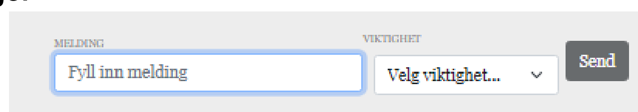
```
deleteMsg() {
  const messageURL = "/message/delete?messageId=" + this.message.messageId;

  AuthoService.delete(messageURL)
    .then(() => {
      this.$emit("deleteMessageAcknowledged");
    })
    .catch(reason => {
      this.error = reason;
      console.log(reason);
    });
},
```

Figur 102 - Implementasjon av deleteMsg-funksjon

I denne funksjonen blir det generert en URL som har som basis URL-en for å slette melding, i tillegg blir det lagt med et «RequestParam» som inneholder IDen til meldingen som skal bli slettet. Dette gjøres enkelt ved at basert på måten meldingene blir lastet inn i hver sin melding vil man enkelt kunne plukke ut IDen til meldingen basert på kortet som blir trykket på. Dette resulterer da til sammen URL-en som skal bli sendt til backenden for å få slettet meldingen. Her benyttes ikke **GameService.js** som har blitt

benyttet tidligere for kall mot backenden. Ettersom dette er et kall som krever autorisering via. JWT token på backenden er vi nødt til å sende med tokenet i headeren til forespørselen, noe vi gjør ved å benytte **AuthoService.js** klassen, som forklart i dypere detalj i et punkt lenger opp. Forespørselen blir da til sutt sendt og om den går gjennom OK som forventet blir meldingen fjernet fra listen.

Sending av meldinger

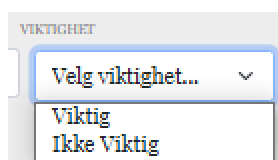


Figur 103 - Skjema for sending av meldinger

En arrangør kan i dashbordet sende meldinger. Dette gjøres ved å fylle ut de nødvendige boksene med ønsket informasjon. Det er to bokser som må fylles ut, her vist i figur 103.

Input-et som må gis er:

- Melding - Hva er meldingen?
- Viktighet - Viktig eller ikke viktig?

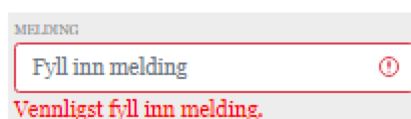


Figur 104 - Visning av valg av viktighet av melding.

Som en kan se i figur 104, er også viktighet laget til å være en nedfellingsliste, hvor standard valget er satt til å være et ugyldig et. Noe som gjør for et enkelt og intuitivt oppsett for å sende melding. Begge disse boksene blir validert i likhet med de andre tekstboksene i andre skjemaer i prosjektet.

- Melding & Viktighet
    - Kreves - Må fylles inn

Dersom det finnes noe feil med noen av boksene vil knappen for å sende meldingen bli deaktivert, og feilmeldinger hvor feilen ligger blir markert under boksen i likhet med andre i prosjektet. Dette er illustrert i figur 105.



Figur 105 - Eksempel på validering av melding.

```
sendMessages() {
  var data = {
    tournamentId: this.tournamentid,
    message: this.v$.message.$model,
    importance: this.v$.importance.$model
  };
```

Figur 106 - Implementasjon av sendMessage-funksjon, del 1

Når de har fylt ut som ønsket og trykker på send knappen blir *sendMessages()* funksjonen kalt. Funksjonen er vist i figur 106. Her blir først dataen fra de utfylte boksene satt inn i et JSON-objekt. Turnerings ID blir ikke fylt inn manuelt, men kan bli hentet ut av seg selv på samme måte som meldings ID ble på sletting av meldinger.

```
  this.v$.$validate();
  if (!this.v$.error) {
    const messagesURL = "/message";

    AuthoService.post(messagesURL, data)
      .then(() => {
        this.$emit("messageSentAcknowledged");
        this.v$.message.$model = "";
      })
      .catch(error => {
        console.log(error);
        if (error.response.status == "401") {
          console.log(
            "User not authorized to send messages to this tournament"
          );
        }
      });
  } else {
    alert("Melding kunne ikke bli sendt.");
  }
}
```

Figur 107 - Implementasjon av sendMessage-funksjon, del 2

Videre, vist i figur 107, blir det sjekket om valideringen er gjennomført OK, dersom den er det blir det gått videre og kallet blir bygget opp på samme måte som sletting av meldinger, ved hjelp av **AuthoService**, ettersom det kreves autorisering for å sende meldinger. Dersom meldingen går gjennom til backenden og responsen er OK blir feltene klarert og brukeren vil kunne få se meldingen sin i listen over meldinger. Dersom det er noe feil, f.eks. med autorisering eller lignende til de spesifikt få beskjed om dette.

Slette turneringen

Dersom en eier av turneringen ønsker å slette turneringen, kan de få gjøre dette. Det er laget en egen knapp for dette som en del av dashbordet til turneringen. Seksjonen med knappen for dette er igjen også bak en «v-if:» sjekk, denne er igjen den nevnt lenger oppe hvor den sjekker om bruker IDen til innloggede bruker og IDen til eieren av turneringen stemmer, dersom den gjør det vil knappen bli lastet inn og er tilgjengelig for brukeren. Når knappen blir trykket på kaller det funksjonen *deleteTournament.* Denne funksjonen er vist i figur 108.

```
deleteTournament() {
  const deleteTournamentURL = "/tournaments/delete?tournamentid=" + this.id;
  AuthoService.delete(deleteTournamentURL)
    .then(() => {
      this.$router.go(-1);
    })
    .catch(error => {
      this.error = error;
      console.log(this.error);
      this.errorFeedBackText = "Kunne ikke slette turnering.";
    });
}
```

Figur 108 - Implementasjon av deleteTournament

Denne funksjonen tar da og bygger opp en URL og henter ut IDen til turneringen den er i og fullfører URL-en med denne. Deretter benytter denne også **AutoService.js** som følge av kravet om autorisering. Dersom forespørselen går gjennom vil brukeren via Vue sin ruter bli sendt tilbake et hakk, som vil være profilen deres.

4.6.2.11 Administratorpanel

Figur 109 - Skjema for endring av en brukers rolle

Nevnt i punkt ovenfor har administratorer på sin profil en knapp tilgjengelig hvor de ved trykk blir sendt til et eget Administratorpanel. I dette Administratorpanelet, vist i figur 109, kan Administratorbrukere gjennomføre endringen av rollen til en gitt bruker. I likhet med dashboard for turnering blir det gjennomført en sjekk om hvilken rolle en bruker har når siden blir lastet inn, vist i figur 110.

```
<div v-if="gettersAuthData.role === 'ROLE_ADMIN'">
```

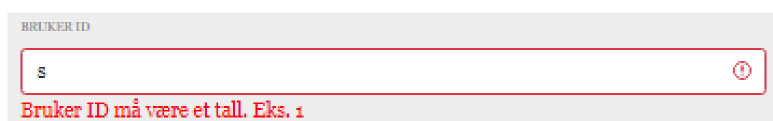
Figur 110 – Eksempel på betinget lasting av element

Her benyttes **gettersAuthData** for å hente ut authData objektet og deretter blir det sjekket om rolle elementet i dette objektet er den nødvendige rollen, "ROLE_ADMIN". Dersom den er det vil div boksen med panelet bli lastet inn, og hvis det ikke er det blir en annen boks med tilbakemelding om manglende autorisering lastet inn.

I skjemaet er det to tekstbokser som skal måtte bli fylt ut. Første er hvor det skal fylles inn, og andre er hvilken rolle den skal måtte bli endret til. Disse to boksene blir i likhet med de andre tekstboksene i prosjektet validert. Følgende validering blir gjort på boksene:

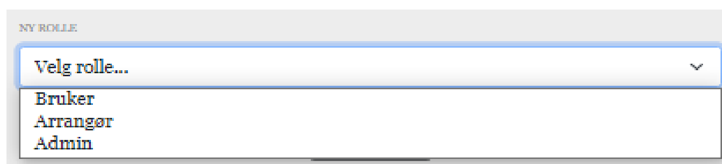
- Bruker ID
  - Kreves - Må fylles inn
  - Numerisk - Må være et tall
- Rolle
  - Kreves - Må fylles inn
  - Numerisk - Må være et tall
  - Mellom x og y - Må være et tall mellom 1 og 3.

Dersom innholdet i tekstboksen for bruker ID blir fylt ut med noe annet enn et tall vil det komme en feilmelding som spesifiserer dette, vist i figur 111.



Figur 111 - Eksempel på validering i felt for bruker-id

Feltet for å velge hvilken rolle brukeren skal få gjøres ved en nedfellingsliste, vist i figur 112.



Figur 112 - Eksempel på valg av ny rolle til bruker

Her er standardvalget satt til "velg rolle.." som er et avslått valg som gjør at brukere ikke med et uhell kan komme borti send og endre rollen til en bruker til noe uønsket dersom de skulle trykke noe feil etter de har fylt ut bruker ID. Vi kan se at i denne listen er det en liste over navnene til rollen, men på backenden, og i valideringen er det spesifisert at dette må være visse tall. Dette gjøres ved å spesifisere verdien til valgene i denne listen, som vist i figur 113.

```
<option value="" disabled selected hidden>Velg rolle...</option>
<option value="1">Bruker</option>
<option value="2">Arrangør</option>
<option value="3">Admin</option>
```

Figur 113 - Visning av mapping mellom verdier og tekst gjennom et option-element

Her kan man se at dersom "bruker" blir valgt, vil verdien til valget være "1" som er den verdien som blir hentet ut av boksen og hentet inn i forespørselen. Når Administratorbrukeren har fylt inn nødvendig data og trykker send vil en funksjon bli kjørt, *updateRole***.**

```
updateRole() {
  this.v$.$validate(); // checks all inputs
  if (!this.v$.$error) {
```

Figur 114 - Implementasjon av updateRole-funksjonen, del 1.

I figur 114 ser man at det først gjennomført en sjekk om det er noe feil med valideringen, dersom det er det vil forespørselen ikke gå lenger og blir avbrutt og et varsel om dette blir sendt til brukerens nettleser.

```
const url =
  "/auth/updateRole?userId=" +
  this.v$.userId.$model +
  "&roleId=" +
  this.v$.roleId.$model;
```

Figur 115 - Implementasjon av updateRole-funksjonen, del 2

Videre, i figur 115, blir det bygget en URL som er hvor forespørselen skal bli sendt. Her har det blitt definert en grunn URL som trenger to parameter. Disse to blir hentet ut fra den utfylte dataen i skjemaet.

```
AuthoService.put(url)
  .then(() => {
    alert("Rolle endret");
  })
  .catch(error => {
    this.error = "error";
    if (error.response.status == 403) {
      alert("Kan ikke bytte din egen rolle");
    }
    console.log(error);
  });
```

Figur 116 - Implementasjon av updateRole-funksjonen, del 3

Etter dette, vist i figur 116, blir det hentet inn **AuthoService** for å sende URL, denne blir benyttet her ettersom det kreves autorisering for å gjennomføre denne handlingen. Her blir det da benyttet *put()* funksjonen i denne klassen, dette fordi det er en PUT forespørsel som blir sendt da det skal gjennomføres en endring. Her blir da URL-en som ble bygget sendt med som en parameter og forespørselen blir sendt. Dersom brukeren har forsøkt å endre sin egen rolle vil det bli mottatt en error som spesifiserer dette. Dersom alt gikk gjennom OK, vil brukeren bli informert om dette og en bruker sin rolle har da blitt endret.

5 DRØFTING

I dette kapittelet tar gruppen for seg drøfting av tekniske resultat, organisering, planlegging av prosjektet og forbedringer på tjenesten.

5.1 Teknisk resultat

Dette delkapittelet tar for seg en drøfting rundt det tekniske resultatet av løsningen.

5.1.1 Hele løsningen

Gruppen er generelt fornøyd med hva de har fått til. Løsningen løser flere punkter i problemstillingen. Det mangler derimot en del funksjonalitet gruppen mener de kunne ha fått til hadde gruppen planlagt tiden bedre. Løsningen henter ikke noe data fra noen turnering. Ei heller har ikke løsningen noen av delen rundt turneringsinformasjon. Den viser derimot at man får videresendt data fra backend til frontend og presentert dataen på et greit vis.

5.1.2 Frontend

Dette delkapittelet tar for seg en drøfting rundt det tekniske resultatet av løsningen, for frontend.

5.1.2.1 Generell

Frontend utvikling er noe gruppen ikke hadde veldig mye erfaring med i forkant av prosjektet. Noe som gjorde at det gikk med mye tid på å sette seg inn og forstå hvordan dette fungerte. I starten var det veldig mye spørsmål og usikkerhet med hvordan noe gjøres, etter hvert som gruppen fikk erfaring med det som ble brukt i frontenden gikk ting veldig greit og sto for en generell grei gjennomførelse. Gruppen føler de har fått en tjeneste som gir en god brukeropplevelse for det som er til stede.

5.1.2.2 SPA, CSR og Single File Components

Gruppen er fornøyd med valget av å implementere en Single Page Application i Vue. Konseptet med Single File Components lettere for en utvikler å utvikle komponenter, da alt man trenger å styre presentasjonen kan gjøres i en og samme fil.
Samtidig gir Client Side Routing i en Single Page Application en myk opplevelse for brukeren, i og med man slipper å laste inn sider. Det gir mer kompleksitet i presentasjonslaget, men det føles vel verdt.

5.1.2.3 Valg av Vue versjon

Under oppsett av Vue prosjekt får man et valg om hvilken versjon av Vue man ønsker å benytte, Vue 2 eller Vue 3. Når gruppen opprettet prosjektet her, ble det valgt Vue 3. Dette var et valg som endte opp med å gå imot gruppen til en stor grad. Vue 3 er en ny versjon av Vue som ble gitt ut i slutten av 2020. Som følge av en stor grad av redesign i den fundamentale strukturen i hvordan rammeverket fungerer resulterte dette i større problemer. Vue 2 er veldig populært og brukt mye, som har gitt et stort fellesskap rundt det som gjør det lett å finne mulige løsninger på problemer, eller idéer for hvordan man skal gjøre en funksjonalitet. Som følge av Vue 3 sin korte tid ute gjorde det at det var

veldig vanskelig å finne hjelp dersom man sto fast på noe da alt man fant var for Vue 2, og disse løsningene fungerte sjeldent i Vue 3.

Ytterligere problem kom når gruppen trengte diverse pakker eller rammeverk i prosjektet. Her var det veldig stor støtte for Vue 2, noe som ikke kunne bli sagt om Vue 3. Det fantes ekstremt lite nyttige rammeverk for Vue 3, noe som resulterte i at gruppen hadde et veldig dårlig utvalg av hva som skulle bli benyttet i prosjektet.

Dette var et valg som kom som følge av mangel på ordentlig undersøking innenfor de spesifikke versjonene av Vue. Da gruppen realiserte problemet var vi allerede for langt ute i det til at det var mulig å gjøre om vil Vue 2. Dersom gruppen skulle gjort det samme igjen ville den ha valgt Vue 2 av disse årsakene.

5.1.3 Back-end

Dette delkapittelet tar for seg en drøfting rundt det tekniske resultatet av løsningen, for backend.

5.1.3.1 Generell

Backenden var det første som ble fokusert på under utviklingen. Ettersom mye av studiet og siste emnene gruppemedlemmene har hatt har vært fokusert på backend utvikling med Java sentralt hadde gruppen et generelt grunnlag for å kunne sette i gang med dette rimelig fort. I denne løsningen benyttet gruppen et annet rammeverk, Spring, i oppgaven enn vi har drevet med tidligere, Java 8 med Jakarta EE-implementasjon. Det gav til tider noen spørsmål og usikkerhet vedrørende hvordan dette skulle gjøres i Spring i motsetning til slik vi hadde gjort det tidligere med Jakarta EE. Fordelen med Spring er at det er et ekstremt populært rammeverk, som gjorde at å finne løsningen og hjelp på problemer ikke var veldig vanskelig da det er mye ressurser tilgjengelig for å kunne hjelpe en videre.

5.1.3.2 Arkitektur

Valget av en lagvis arkitektur fungerte bra for gruppen. Det gav koden en fin logisk atskillelse. Denne arkitekturen er lett gjennomførbart med de forskjellige Spring-rammeverkene og deres annoteringer.

5.1.3.3 REST-API er ikke REST-API

Gruppen sin implementasjon av et REST-API følger ikke REST-arkitekturstilen helt ut, så systemet kan ikke kalles et RESTful system. Vår implementasjon bryter med prinsippene om et uniformt grensesnitt og at det ikke er støtte for Caching. Hvor det brytes når det gjelder uniformt grensesnitt, er at ingen responser inneholder hypertekstlenker til andre ressurser. Fraværet av disse hypertekstlenkene gjør at det blir en tettere knytning mellom frontend og backend enn nødvendig. Slik det er nå må klienten vite eksakt rute til de fleste ressurser, i stedet for å bli informert av server om hvilke ressurser finnes, og hvor ressursene finnes, gjennom hypertekstlenker i responsene.

5.1.4 Database

Databasen ble ikke planlagt i nøye detalj i forkant av gjennomførelsen, databasen ble sakte, men sikkert utvidet og formet sammen etter hvert som utviklingen skjedde og

flere enheter ble lagt inn i systemet og som trengte en tabell og diverse relasjoner. Etter hvert som utviklingen gikk videre ble det mer og mer nødvendig å se dypere på konkrete relasjoner imellom de forskjellige tabellene ettersom det oppsto situasjoner hvor måten vi ønsket å gjennomføre påkrevde spesifikke oppsett på database siden.

Selve databasen ble satt opp av Spring under oppstarten, noe som gjorde den veldig enkel å administrere da en dermed slapp å måtte kjøre et slags egendefinert skript. Spring så på hvordan enheter og lignende var definert, samt relasjoner og mer og satte opp databasen basert på dette, og fremt det ikke var noen problemer var databasen satt opp som ønsket når backend serveren var oppe.

5.2 Gjennomførelse av prosjektet

Dette delkapittelet tar for seg en drøfting av gjennomførelsen av prosjektet.

5.2.1 Organisering

Som nevnt under 3.1.1, hadde gruppen ingen utvalgt leder. Gruppen var mindre vant med å jobbe med hverandre i et så stort prosjekt og oppgaven føltes overveldende. Denne kombinasjonen førte til at det i starten av prosjektet var mindre vilje til å dra i gang arbeid. Hadde det vært utnevnt en leder så tror gruppen at dette hadde ført til at personen hadde hatt mer autoritet til å stake ut en retning i starten.

5.2.2 Prosjektstyring

Gruppen prøvde seg i starten på å styre prosjektet med å bruke bare Discord og Google Docs. Dette viste seg raskt at bare de to tjenesten ikke fylte behovet man har når man skulle starte utvikling og jobbe etter Scrum. Gruppen gikk så over til å ta i bruk Jira og Confluence, men beholdt Google Docs for lagring av dokumenter som ikke var prosjektstyringsrelatert.

5.2.3 Scrum

I de tidligste utviklingssprintene var utførelsen av Scrum mindre god. Det var ingen som fylte rollene, og det ble lagt mindre vekt på møtene, da særlig planleggingsmøtene. Dette førte til at det var litt tilfeldig hva som kom inn på sprintene og de punktene ble om oftest helt feilestimert. Dette forbedret seg derimot kraftig. Gruppen startet å diskutere planlegge arbeidsoppgavene mye mer nøye og ene medlemmet tok på seg rollen som produkteier. Dette ble og hjulpet av at gruppen gikk bort fra avtalen om individuell arbeidsdag på fredager, nevnt i forrige punkt, slik gruppen fikk bedre tid til planlegging.

5.3 Planlegging og tidsbruk

Gruppen har gjennom hele prosjektet ikke hatt en satt tidsplan. Det førte til at gruppen ikke alltid var klar over hvor de lå an i løypa med tanke på arbeids om måtte utføres. Til slutt ble vår løsning på problemstilling skadelidende.

I forprosjektet beskrev gruppen en liste over hovedaktiviteter i prosjektet og det ville vært naturlig å oversette disse aktivitetene til en kalender et Gannt-diagram. En slik

visuell representasjon med tidsperioder og datoer kunne gjort gruppen klar over hvordan gruppen lå an.

5.3.1 Periode med Informasjonsinnsamling

Et av det første aktivitetene gruppen gjorde var å utføre en periode med informasjonsinnsamling. Gruppen undersøkte eksisterende løsninger innen web sjakk og språk og rammeverk. Det omhandlet for det meste å få en idé om hva som eksisterte innenfor det generelle området som gruppen hadde som oppgave, i tillegg til hva slags språk og rammeverk som egnet seg best for oppgaven.

Innenfor web sjakk fikk gruppen et innblikk i hvordan oppsettet til nettsidene hos allerede eksisterende løsninger så ut, noe som var til hjelp når gruppen skulle designe utseende på frontend. Det ble også lagt et fokus på å lære terminologi brukt i sjakk-/web sjakk-verdenen, som da kunne brukes i gruppas eget prosjekt. Dette ble da ikke brukt til en særlig stor grad, og kan derfor bli sett på som sløsing av tid i retrospekt.

Når gruppen uken deretter så på programmeringsspråk og rammeverk. Gruppen hadde allerede en viss idé om hva som skulle brukes, men å velge rett språk og rammeverk til oppgaven er viktig lærdom å ta med seg videre. Det var spesielt viktig å finne ut av hvilket rammeverk som gruppen skulle ende opp med å bruke på frontend til slutt. Det ble valgt å gå med en kombinasjon av Java + Spring Boot + Tomcat for backend, og JavaScript + Vue.js for frontend.

Etter perioden var over, var gruppen etter hvert misfornøyd med tidsbruken. Arbeidet tok opptil 9 arbeidsdager, når det ikke hadde trengt noe mer enn kanskje maks en par dager. Gruppen visste allerede at det fantes løsninger vedlagt fra oppdragsgiver som ble laget i Spring og i Vue.js, så det hadde nok ikke trengt å ta så lang tid som det gjorde å finne ut av hva gruppen ville gå videre med.

Alt i alt var undersøkelsene som gruppen gjorde for det meste verdifulle i det stadiet av prosjektet som gruppen var i. Hovedproblemet var tidsbruken, og muligens elementer av undersøkinga som ikke ble brukt videre. Men det hadde vært vanskelig å vite på forhånd.

5.3.2 Underestimering av tid

I starten tenkte gruppen at som den virket som det kunne være lite å gjøre, samtidig som at det var mye potensiale for å kunne øke takhøyden i form av ekstra funksjonalitet og mer. Dette viste seg å være en sterk underestimering av gruppen da det viste seg at oppgaven var mer krevende enn vi hadde sett for oss. På grunn av denne underestimering gav det gruppen en følelse av å ha god tid, noe gruppen egentlig ikke hadde.

5.3.3 Sen start av utvikling

Gruppen kom ikke fullt i gang med utviklingen før i uke 10. Dette var unødvendig sent. Det var flere grunner for dette.

En grunn er at gruppen kom altfor sent i gang med å produsere design artefakter, slik som mockups av nettsiden, sidekart, osv. Gruppen fikk ikke dannet seg en felles

forståelse av hva prosjektet innebar. Det førte til at gruppen i en liten periode famlet rundt uten et felles mål og retning for prosjektet.

En annen grunn avtalen som ble gjort for hvilke dager gruppen skulle møtets. De første gruppen gjorde etter å ha levert forprosjektet var å gjøre noen undersøkelser av sjakk-tjenester og språk og rammeverk. Grunnet at gruppen avtalte at vi møttes først etter forelesninger, ville avslutte dagene seneste klokken 16, og at fredager skulle være en individuell arbeidsdag, så gav det lite fleksibilitet til hvilke dager gruppen skulle avslutte disse undersøkelsene. Derav endte det med at disse to undersøkelsene tok, som nevnt i punkt ovenfor, 9 arbeidsdager å få gjennomført. I samme periode hadde ikke gruppen definert noen andre arbeidsoppgaver, så det var alt vi gjorde disse 9 dagene. Det at fredager var en individuell arbeidsdag gav også utfordringer for starten av sprinter. Vi avsluttet og startet nye sprinter torsdager, samme dagene vi har møter oppdragsgiver og veileder. I tillegg til en forelesing samme dag, så gav dette lite tid til å gjøre god planlegging for sprintene. Da når fredager ikke var tilgjengelig for gruppen og møtets, så ble det slik at sprintene ble underplanlagt.

Etter hvert ble denne avtalen individuell arbeidsdag sett bort fra og gruppen møtets også på fredager og produktivitet økte.

5.4 Forbedringer og utvidelser

Hadde gruppen hatt mer tid er de følgende punktene de vi ville fokusert på å utvikle.

5.4.1 Manglende konsepter

Gruppen føler de mangler noen essensielle deler av løsningen. Gruppen traff ikke helt med forståelsen hvordan en turnering fungerer. Løsningen mangler konsepter som at turneringer har runder, og at turneringer også kan utføres av klubber og ikke bare personer.

Samtidig, for grunner nevnt i punkter om planlegging og tidsbruk, så rakk ikke gruppen å gjennomføre delene for informasjonshenting av turneringsinformasjon og visning av turneringsinformasjon, slik som resultatlister og rundeoversikter. Denne delen skulle opprinnelig tas fra en løsning oppdragsgiver selv hadde, da utrekningen av resultatlister er temmelig kompleks. Det som hindret å bruke denne var at løsningen fulgte en annen arkitektur, som gjorde det utfordrende å føre dette over. Gruppen fikk da ikke tid til å innføre denne delen i løsningen.

Her følger en ikke utfyllende liste over ting som vi kunne tenkt å legge til:

- Klubber
  - Knytte klubber til turnering ettersom det oftest er klubber som arrangerer turneringene.
- Runder i turneringer.
  - Turneringer skjer i runder. Vi har ingen runder støttet.
- Turneringsinformasjon
  - Ingen resultatlister eller rundeoversikter.
  - Gi seere mer informasjon om sjakkpartier
    - Sjakkåpninger
    - Siste trekk
    - …med mer

- Henting av informasjon fra turneringer.

5.4.2 Bruke Sockets for turneringer

I og med løsningen skal gjengi informasjon fra turneringer, hvor noe av denne informasjonen kan bli oppdateres ofte, ville det vært en forbedring for løsningen å bruke sockets og så strømmet dataen til presentasjonslaget. Dette ville gitt flere fordeler. Oppdateringene kunne blitt sendt når de skjer til presentasjonslaget i stedet for å bli hentet på et fast intervall med en REST-API. Dette ville gått raskere og redusert nødvendige HTTP-kall til backend kraftig.

5.4.3 Implementere et faktisk REST-API

Som beskrevet i 5.9.4, så er faktisk ikke API-et til backenden et REST-API. Rammeverket Spring har en modul kalt Sprint HATEOAS [95]. Denne modulen følger spesifikasjonen JSON Hypertext Application Language, HAL [96], som definerer en standard for å representere hypertekstlenker i JSON. Ved å ta i bruk denne modulen ville man kunne enklere implementere REST-arkitekturstilen.

5.4.4 Ny arkitekturer for løsningen

Gruppen har sett for seg hvordan de ville ha gjort om arkitekturen for løsningen. Tiltenkt arkitektur ligger som vedlegg K.

6 KONKLUSJON

6.1 TEKNISK RESULTAT

Aalesund Schaklag sin hovedproblemstilling var at informasjon rundt og fra sjakkturneringer var lite strukturert. Oppgavebeskrivelsen inneholdt mange mulige idéer og ønsker til en løsning. I med oppdragsgiver kom de frem at det var opp til gruppen å velge vinklingen til oppgaven. Valget falt på å utvikle en informasjonsskjerm som kunne brukes ved turneringer. Dette utartet seg til en fullstack løsning med en nettside som frontend.

Valgene av teknologier, rammeverk og språk har gruppen vært generelt fornøyde med.

Frontend skulle være en Single Page Application, som kunne brukes i sjakkturneringer til å samtidig vise informasjon fra en sjakkturnering og hvor turneringsledelse kunne sende informasjon til. Gruppen var fornøyd med valget av rammeverket Vue, som i seg selv var en veldig god opplevelse å arbeide med. Gruppen var mindre fornøyd med valget av versjonen av Vue. Problemet var, som nevnt i rapporten, at under oppsettet ble det valgt å benytte versjonen Vue 3 istedenfor Vue 2. Det ble et problem for bruk av biblioteker og rammeverker, da støtten for Vue 3 var minimal, samt informasjon og hjelp til feilsøking var vanskeligere å komme over.

For backend ble det utviklet en REST-API ved hjelp av Java og Spring-rammeverket. Dette var et valg som viste seg å være veldig gunstig. Gruppen hadde erfaring med Java fra tidligere, men var nye for Spring. Spring viste seg å være et rammeverk som var behagelig og raskt å arbeide med og ga ingen spesielle utfordringer for gruppen. Arkitekturen for backend-løsningen ble en lagvisarkitektur som det var god støtte for i Spring. Samtidig fant gruppen ut underveis, som nevnt i diskusjon, at det vi kalte en REST-API, var ikke helt en REST-API.

Gruppen endte opp med å måtte kutte noe av planlagt funksjonalitet. Løsningen mangler funksjonaliteten for å hente informasjon fra programvaren brukt ved sjakkturneringer, samt har ikke mulighet til å vise resultater, rundeoversikt. Den mangler og noen konsepter som er relevante til sjakkturneringer, slik som at turneringer har runder og det som oftest klubber, ikke personer som arrangerer turneringene. I tillegg ville gruppen ha gjort mer av kommunikasjonen mellom backend og frontend til å bruke stream og sockets enn et REST-API, da med tanke på hyppig oppdatert turneringsinformasjon.

Når det er sagt, så er produktet som har blitt utviklet, noe gruppen er fornøyde med. Vi har fått utviklet en løsning som gjennomfører mye av det ønskede. Vi har fått forsøkt oss på å arbeide med nye teknologier vi ikke har jobbet med før. Å være involvert i et prosjekt hvor man utvikler et produkt som en kunde har forespurt er god erfaring å ha med videre i arbeidslivet.

6.2 GJENNOMFØRELSE AV OPPGAVEN

Gruppen fulgte en Smidig utviklingsmetodikk, Scrum. Gruppen hadde noen problemer i starten med å følge metodikken, men dette bedret seg gjennom prosjektet.

Planlegging er gruppens største forbedringspunkt. Noe planlegging ble utført bra, slik som kravspesifikasjoner og undersøkelser, men gruppen mistet oversikt over tidsbruk og kom sent i gang med selve utviklingen. Når gruppen endelig var varme i trøyen, så var det tid til å avslutte. Gruppen er sikre på at de kunne løst mye mer av oppgaven, om utviklingen hadde startet tidligere.

Oppgaven har vært lærerik for alle gruppemedlemmene. Vi har fått forsøkt oss på å utvikle et produkt i fullstack og har fått fulgt prosjektet fra ide til produkt. I løpet av denne tiden har vi fått erfaring med å ikke bare gjennomføre selve utviklingen, men også gjennomføre det prosjekttekniske. Dette har vært veldig gunstig erfaring å ha med seg videre ut i arbeidslivet.

7 REFERANSER

[1] Atlassian, «Gitflow Workflow,» [Internett]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[2] Wikipedia, «Agile Software Development,» [Internett]. Available: https://en.wikipedia.org/wiki/Agile_software_development.

[3] «agilemanifesto,» [Internett]. Available: http://agilemanifesto.org/.

[4] «12 Principles behind the agile manifesto,» [Internett]. Available: https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/.

[5] Wikipedia, «Scrum,» [Internett]. Available: https://en.wikipedia.org/wiki/Scrum_(software_development).

[6] Atlassian, «Scrum Roles,» [Internett]. Available: https://www.atlassian.com/agile/scrum/roles.

[7] Atlassian, «Scrum Artifacts,» [Internett]. Available: https://www.atlassian.com/agile/scrum/artifacts.

[8] Atlassian, «Scrum Backlogs,» [Internett]. Available: https://www.atlassian.com/agile/scrum/backlogs.

[9] Atlassian, «Scrum Sprint Planning,» [Internett]. Available: https://www.atlassian.com/agile/scrum/sprint-planning.

[10] scrumandkanban, «Sprint Reports,» [Internett]. Available: https://scrumandkanban.co.uk/sprint-reports/.

[11] Wikipedia, «Kanban,» [Internett]. Available: https://en.wikipedia.org/wiki/Kanban_(development).

[12] Atlassian, «Kanban Boards,» [Internett]. Available: https://www.atlassian.com/agile/kanban/boards.

[13] Atlassian, «User Stories,» [Internett]. Available: https://www.atlassian.com/agile/project-management/user-stories.

[14] Atlassian, «Epics,» [Internett]. Available: https://www.atlassian.com/agile/project-management/epics.

[15] Atlassian, «Estimation,» [Internett]. Available: https://www.atlassian.com/agile/project-management/estimation.

[16] Google, «Rendering on the web,» [Internett]. Available: https://developers.google.com/web/updates/2019/02/rendering-on-the-web.

[17] Wikipedia, «Single-Page Application,» [Internett]. Available: https://en.wikipedia.org/wiki/Single-page_application.

[18] Wikipedia, «Separation of Concerns,» [Internett]. Available: https://en.wikipedia.org/wiki/Separation_of_concerns.

[19] Dev IQ, «Separation of Concerns,» [Internett]. Available: https://deviq.com/principles/separation-of-concerns.

[20] Vue, «Single-File Component,» [Internett]. Available: https://v3.vuejs.org/guide/single-file-component.html.

[21] Oracle, «Service Layered Architecture,» [Internett]. Available: https://docs.oracle.com/cd/E93130_01/service_layer/service%20layer%20API/Content/Service%20Layer%20Architecture/Service%20Layer%20Architecture.htm.

[22] O'Reilly, «Software Architecture Patterns,» [Internett]. Available: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html.

[23] Wikipedia, «Representational state transfer,» [Internett]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer.

[24] CodecAdemy, «What is rest,» [Internett]. Available: https://www.codecademy.com/articles/what-is-rest.

[25] CodecAcademy, «MVC,» [Internett]. Available: https://www.codecademy.com/articles/mvc.

[26] UiO, «MVC,» [Internett]. Available: https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html.

[27] CodingHorror, «Understanding model view controller,» [Internett]. Available: https://blog.codinghorror.com/understanding-model-view-controller/.

[28] Wikipedia, «Dependency Injection,» [Internett]. Available: https://en.wikipedia.org/wiki/Dependency_injection.

[29] Wikipedia, «Inversion of Control,» [Internett]. Available: https://en.wikipedia.org/wiki/Inversion_of_control.

[30] Spring , «The IoC Container,» [Internett]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html.

[31] Baeldung, «Spring Autowire,» [Internett]. Available: https://www.baeldung.com/spring-autowire.

[32] Spring, «Container Overview,» [Internett]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-basics.

[33] Baeldung, «Spring Bean,» [Internett]. Available: https://www.baeldung.com/spring-bean.

[34] Vue, «What is reactivity,» [Internett]. Available: https://v3.vuejs.org/guide/reactivity.html#what-is-reactivity.

[35] Vue, «Template Syntax,» [Internett]. Available: https://v3.vuejs.org/guide/template-syntax.html#template-syntax.

[36] Vue, «Composing with components,» [Internett]. Available: https://v3.vuejs.org/guide/introduction.html#composing-with-components.

[37] Vue, «The Root Component,» [Internett]. Available: https://v3.vuejs.org/guide/instance.html#the-root-component.

[38] Vue, «Single File Component,» [Internett]. Available: https://v3.vuejs.org/guide/single-file-component.html.

[39] Vue, «Lifecycle Diagram,» [Internett]. Available: https://v3.vuejs.org/guide/instance.html#lifecycle-diagram.

[40] JWT IO, «JWT Introduction,» [Internett]. Available: https://jwt.io/introduction.

[41] Wikipedia, «HTTPS,» [Internett]. Available: https://en.wikipedia.org/wiki/HTTPS.

[42] SNL, «Relasjonsdatabase,» [Internett]. Available: https://snl.no/relasjonsdatabase.

[43] Wikipedia, «Relation Database,» [Internett]. Available: https://en.wikipedia.org/wiki/Relational_database.

[44] Wikipedia, «Primary Key,» [Internett]. Available: https://en.wikipedia.org/wiki/Primary_key.

[45] Wikipedia, «Foreign Key,» [Internett]. Available: https://en.wikipedia.org/wiki/Foreign_key.

[46] Wikipedia, «One-to-one,» [Internett]. Available: https://en.wikipedia.org/wiki/One-to-one_(data_model).

[47] Wikipedia, «One-to-many,» [Internett]. Available: https://en.wikipedia.org/wiki/One-to-many_(data_model).

[48] Wikipedia, «Many-to-many,» [Internett]. Available: https://en.wikipedia.org/wiki/Many-to-many_(data_model).

[49] Wikipedia, «SQL,» [Internett]. Available: https://en.wikipedia.org/wiki/SQL .

[50] Wikipedia, «Object Relational Mapping,» [Internett]. Available: https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping.

[51] Wikipedia, «Portable Game Notation,» [Internett]. Available: https://en.wikipedia.org/wiki/Portable_Game_Notation.

[52] Forsyth Edwards Notation, [Internett]. Available: https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation.

[53] Spring, «Spring Framework,» [Internett]. Available: https://spring.io/projects/spring-framework.

[54] Spring, «Using boot starter,» [Internett]. Available: https://docs.spring.io/spring-boot/docs/2.4.4/reference/htmlsingle/#using-boot-starter.

[55] Spring, «Spring Boot,» [Internett]. Available: https://spring.io/projects/spring-boot .

[56] Spring, «Spring Initializr,» [Internett]. Available: https://start.spring.io/ .

[57] Spring, «Spring MVC,» [Internett]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html.

[58] Spring, «Spring Security,» [Internett]. Available: https://spring.io/projects/spring-security.

[59] Spring, «Spring Security - Exploits,» [Internett]. Available: https://docs.spring.io/spring-security/site/docs/5.4.6/reference/html5/#exploits.

[60] Spring, «Spring Data JPA,» [Internett]. Available: https://spring.io/projects/spring-data-jpa .

[61] Infoworld, «JPA,» [Internett]. Available: https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html .

[62] Javatpoint, «JPA introduction,» [Internett]. Available: https://www.javatpoint.com/jpa-introduction.

[63] Wikipedia, «Jakarta Persistence,» [Internett]. Available: https://en.wikipedia.org/wiki/Jakarta_Persistence.

[64] Wikipedia, «Hibernate,» [Internett]. Available: https://en.wikipedia.org/wiki/Hibernate_(framework) .

[65] Apache, «Maven,» [Internett]. Available: https://maven.apache.org/.

[66] Geeksforgeeks, «Introduction apache maven automation tool,» [Internett]. Available: https://www.geeksforgeeks.org/introduction-apache-maven-build-automation-tool-java-projects/.

[67] Vue, «Vue Introduction,» [Internett]. Available: https://v3.vuejs.org/guide/introduction.html.

[68] ChessJS, «ChessJS Documentation,» [Internett]. Available: https://github.com/jhlywa/chess.js/blob/master/README.md.

[69] Chessboard JS, «Chessboard JS,» [Internett]. Available: https://chessboardjs.com/.

[70] Chessboard JS, «Chessboard JS Repository Github,» [Internett]. Available: https://github.com/oakmac/chessboardjs/.

[71] MinimalGhost, «What is AxiosJS and why should I care,» [Internett]. Available: https://medium.com/@MinimalGhost/what-is-axios-js-and-why-should-i-care-7eb72b111dc0.

[72] Logrocket, «How to make HTTP requests like a pro with axios,» [Internett]. Available: https://blog.logrocket.com/how-to-make-http-requests-like-a-pro-with-axios/#why.

[73] Zetcode, «Axios,» [Internett]. Available: https://zetcode.com/javascript/axios/.

[74] Bootstrap, «Getbootstrap,» [Internett]. Available: https://getbootstrap.com/.

[75] Wikipedia, «Bootstrap,» [Internett]. Available: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework).

[76] Prettier, «Prettier,» [Internett]. Available: https://prettier.io/.

[77] ESLint, «ESLint,» [Internett]. Available: https://eslint.org/.

[78] Babel, «Babel,» [Internett]. Available: https://babeljs.io/.

[79] Webpack, «Webpack,» [Internett]. Available: https://webpack.js.org/.

[80] Vue, «Vue CLI,» [Internett]. Available: https://cli.vuejs.org/.

[81] Vue, «Vue Router,» [Internett]. Available: https://router.vuejs.org/.

[82] Vueschool, «Vue Router course and resource overview,» [Internett]. Available: https://vueschool.io/lessons/Vue Router-course-and-resource-overview.

[83] Digital Ocean, «How to navigate between views with Vue Router,» [Internett]. Available: https://www.digitalocean.com/community/tutorials/how-to-navigate-between-views-with-Vue Router.

[84] Vue, «Vuex,» [Internett]. Available: https://vuex.vuejs.org/.

[85] Vuelidate, «Vuelidate,» [Internett]. Available: https://vuelidate.js.org/.

[86] Monterail, «Rethinking validations for vue js,» [Internett]. Available: https://www.monterail.com/blog/2016/rethinking-validations-for-vue-js/.

[87] Atlassian, «Jira,» [Internett]. Available: https://www.atlassian.com/software/jira.

[88] Atlassian, «Confluence,» [Internett]. Available: https://www.atlassian.com/software/confluence.

[89] Experience UX, «What is wireframing,» [Internett]. Available: https://www.experienceux.co.uk/faqs/what-is-wireframing/.

[90] Wikipedia, «Website Wireframe,» [Internett]. Available: https://en.wikipedia.org/wiki/Website_wireframe.

[91] Sjakkklubb, «TurneringsService,» [Internett]. Available: http://www.sjakklubb.no/turneringsservice/.

[92] digitalgametechnology, «DGT LiveChess Software,» [Internett]. Available: http://www.digitalgametechnology.com/index.php/products/electronic-boards/serial-tournament/285-dgt-livechess-software13.

[93] Bezkoder, «Spring Boot Security PostGresQL JWT Authentication,» [Internett]. Available: https://bezkoder.com/spring-boot-security-postgresql-jwt-authentication/.

[94] learmoreseekmore, «Vue3 JWT Auth AccessToken,» [Internett]. Available: https://www.learmoreseekmore.com/2020/12/vue3-jwt-auth-accesstoken.html.

[95] Spring, «HATEOAS,» [Internett]. Available: https://spring.io/projects/spring-hateoas .

[96] IETF, «HAL,» [Internett]. Available: https://tools.ietf.org/id/draft-kelly-json-hal-01.html.

VEDLEGG

A. **Sjakkskjerm-backend kildekode:**

   Kildekode -> sjakkbackend.zip

B. **Sjakkskjerm-frontend kildekode:**

   Kildekode -> sjakkfrontend.zip

C. **Wireframes:**

   Wireframes -> Wireframes.pdf

D. **Kravspesifikasjoner:**

   Product Requirements -> Kravspesifikasjon.pdf

E. **Undersøkelse:**

   Undersøkelser -> Undersøkelser.pdf

F. **Sprint retrospektivrapporter:**

   Sprint Retrospectives -> Sprint Retrospectives.pdf

G. **Sprint gjennomgangsrapporter:**

   Sprintrapporter -> Sprintrapporter.pdf

H. **Møtereferat:**

   Møtereferat -> Oppdragsgiver -> Møtereferat - Oppdragsgiver.pdf
   Møtereferat -> Veileder -> Møtereferat - Veileder.pdf

I. **Forprosjektsrapport og presentasjon:**

   Forprosjektrapport -> Forprosjektrapport.pdf

J. **Arbeidslogger:**

   Arbeidslogger -> Arbeidslogger.pdf

K. **Arkitekturforslag:**

   Arkitektur -> ny_arkitektur.png

L. **Databasedesign:**

   Databasedesign -> databasemodell_final.PNG

NTNU
Kunnskap for en bedre verden