LUFTKVALITET I STORBYENE

Hjemmeeksamen IN2000

Patrick Kvistbråten Gove (patrikg)
Patrik Gillebo Olsen (patrikgo)
Johnmar Geli Riiser (johnmarr)
Sindre Sukkestad Gulbrandsen (sindrsg)

Veileder: Isak Forstrøm Fyksen (isakff)

Innholdsfortegnelse

1 Presentasjon

Denne rapporten er utviklet for en hjemmeeksamen i faget *IN2000 - Software Engineering og prosjektarbeid* som blir undervist på Universitetet i Oslo våren 2019. Hjemmeeksamen består av et prosjektarbeid hvor studentene blir delt inn i grupper og skal utvikle et system som skal basere seg på værdata levert fra Metrologisk institutt. Det ble gitt fem forskjellige caser hvor studentgruppene selv skulle velge en case de ville ta for seg. Vår gruppe valgte å gå for case 4: *luftkvalitet i storbyene* ettersom vi følte dette var den oppgaven det ville bli mest spennende å jobbe med, samt vi så for oss flere interessante utfordringer vi ønsket å utvikle løsninger for. Oppgaveteksten lyder som følger:

I vintermånedene hører vi ofte i mediene om at luftkvaliteten i storbyene er dårlig. Tiltak som dieselforbud, piggdekkforbud og rushtidsavgift har vært utprøvd, men dårlig luftkvalitet er tidvis et stort problem likevel. Astmatikere og folk med luftveissykdommer er spesielt hardt rammet når luftkvaliteten er dårlig.

Deres oppgave er i dette prosjektet er å lage en app som visuelt fremstiller luftkvaliteten i Norges byer. Ønsket funksjonalitet kan (men må ikke) være:

- *Fremstilling av luftkvalitet på et kart*
- *Forventet luftkvalitet de kommende dagene gitt værvarsel*
- *Varsling til brukeren når luftkvaliteten i bestemt by er dårligere enn egendefinert verdi.*

Disse er forslag til funksjonalitet. Appen er ikke nødt til å ta for seg disse, og det er fult mulig å være kreativ og lage sine egne varianter/funksjonaliteter utover det som er oppgitt i listen.

Oppgaven er gitt av Universitetet i Oslo i samarbeid med Metrologisk Institutt. Vår løsning består av fremvisning av luftkvalitet på egen lokasjon, samt både funksjoner for visning av luftkvalitet på kart og luftkvalitet på relevante steder for brukeren i form av en liste. Vi har i hovedsak hatt fokus på å fremvise luftkvaliteten på en god og forståelig måte for et bredt spekter av brukere, altså passet på at fremvisningen er enkel og selvforklarende. I tillegg har

vi passet på at brukeren får enkle, men nyttige helsetips med utgangspunkt i luftkvaliteten. Det har også blitt implementert varsling og muligheter for tilpasning for hver enkelt bruker.

1.1 Gruppen

Gruppen vår består av fire medlemmer som alle går studiet *Programmering og Systemarkitektur* ved Universitetet i Oslo. Vi har gruppenummer 09, og har valgt å kalle oss *Grønn Programmering*. Alle gruppemedlemmene er godt kjent med hverandre og vi har siden starten av studietiden samarbeidet på flere prosjekter og oppgaver, og har derfor en del erfaring med samarbeid med de andre gruppemedlemmene. Hele gruppen syntes det er viktig å ta vare på miljøet, derfor ble valget av case påvirket av et ønske fra gruppen om å arbeide for en høyere forståelse av farer ved dårlig luftkvalitet.

1.1.1 Johnmar Geli Riiser

Johnmar Geli Riiser går 4. semester på bachelorgraden Informatikk: Programmering og systemarkitektur. Han har fra tidligere noe erfaring i Python gjennom personlig interesse, men har i løpet av studietiden bygd mye erfaring med programmering. Favorittspråket hans er Python fordi det er allsidig og intuitivt.

1.1.2 Patrick Kvistbråten Gove

Patrick Kvistbråten Gove går 4. semester på bachelorgraden Programmering og Systemarkitektur. Han har i likhet med andre i gruppa mye erfaring med programmering fra både videregående og egen personlige interesse. Ellers er han glad i å utforske ny teknologi og løsninger, og jobbe kreativt. Favorittspråket hans er C, ettersom C er mer prosessorientert, mens Java til sammenlikning er mer dataorientert.

1.1.3 Patrik Gillebo Olsen

Patrik Gillebo Olsen går 4. semester på bachelorgraden Programmering og Systemarkitektur, og har ellers mye erfaring med programmering fra både videregående og av egen personlige interesse. Ellers er han glad i å skrive tekst, og å jobbe kreativt. Favorittprogrammeringsspråket hans er Java, ettersom det kan brukes ved løsning av mange interessante matematiske og logiske oppgaver.

1.1.4 Sindre Sukkestad Gulbrandsen

Sindre Gulbrandsen tar bachelor studiet Programmering og Systemarkitektur på Universitetet i Oslo. Han er nå på sitt 4. semester og tar for tiden IN2000 - Software Engineering med prosjektarbeid og IN2140 - Introduksjon til operativsystemer og datakommunikasjon. Tidligere har erfaring med programmering i Java, Python, C#, C og JavaScript, med litt fokus på applikasjoner i C#, ellers har han mest erfaring med objekt orientert programmering med utskriving i terminal.

2 Brukerdokumentasjon

Systemet skal kjøre på smarttelefoner som kjører Android. Den virker også bare på telefoner som har *API*[1] nivå 26 eller høyere. Det er på grunn av at enkelte funksjoner ikke virker på de eldre versjonene av Android. Dette tilsvarer Android versjon 8.0.0 eller nyere. Dessverre vil færre brukere få tilgang til systemet på grunn av det. Det var et valg vi var nødt til å ta slik at systemet kunne optimaliseres for de som faktisk har mulighet til å bruke det. Spesifikke målgrupper vi har tatt spesielt hensyn til er personer med luftveissykdommer. Vi har derfor inkludert tips til hva man burde gjøre dersom man er spesielt utsatt. Vi har også inkludert en liten guide på hvilke stoffer som kan være ekstra viktig å fokusere på dersom man skulle ha en sykdom. Vi valgte likevel å utvikle et system som den gjennomsnittlige personen enkelt

[1] API (Android Enheter) = Mål for hvilken versjon enheten kjører på

kan ta i bruk selv om man ikke skulle være spesielt interessert i luftkvaliteten rundt om i landet. Språket til systemet når det tas i bruk på en emulator er Norsk, ettersom luftkvaliteten bare måles på stasjoner i Norge. Vi så derfor ingen grunn til å implementere andre språk i systemet.

2.1 Fargekoder

Når det kom til informasjonen vi presenterte var det viktig for oss at brukeren enkelt kunne skille mellom hva som tilsvarte gode og dårlige verdier. Slik trengte man ikke nødvendigvis så mye kunnskap om verken luftkvalitet eller de ulike forurensende stoffene i luften. Vi har derfor fargekodet skyen som presenterer *AQI*-nivå[2], og boksene som inneholder mengden av hvert *forurensende stoff*[3]. Når den momentane luftkvaliteten er bra vil skyen være grønn, om den er middels dårlig vil den bli gul, om den er dårlig blir den rød, og til slutt om den er veldig dårlig blir fargen lilla. Det samme gjelder også infoboksene under. De farges etter hvor store mengder av stoffene som finnes i luften. Denne rangeringen er blitt direkte hentet fra Met som har en satt farge for et spekter av verdier. Siden de har god kunnskap om dette temaet har vi også tatt i bruk deres beskrivelse. Fargekodene som er blitt brukt er universelt forstått og hentes også fra Met. Fargene og rangeringene som vises i tabellen under er helt likt som det systemet bruker. Tabell 2 er fargene som man kan ta i bruk om man er fargeblind.

1 Eksempel på bruk av fargekoder

Rangeringer	**Luftkvalitet (*AQI*)**	**Pm10**	**Pm2.5**	**NO2**	**O3**
Lav	1 - 1.9	0 - 29	0 - 14	0 - 99	0 - 99
Moderat	2 - 2.9	30 - 49	15 - 24	100 - 199	100 - 179
Høy	3 - 3.9	50 - 149	25 - 74	200 - 399	180 - 239
Veldig høy	4+	150+	75+	400+	240+

[2] AQI (air quality index) = Indeks som viser hvor forurenset luften er. Ulike indikatorer rundt om i verden.
[3] Forurensende stoffer = PM10, PM2.5, NO2 og O3

Rangeringer	Luftkvalitet (*AQI*)	Pm10	Pm2.5	NO2	O3
Lav	1 - 1.9	0 - 29	0 - 14	0 - 99	0 - 99
Moderat	2 - 2.9	30 - 49	15 - 24	100 - 199	100 - 179
Høy	3 - 3.9	50 - 149	25 - 74	200 - 399	180 - 239
Veldig høy	4+	150+	75+	400+	240+

2.2 Min lokasjon

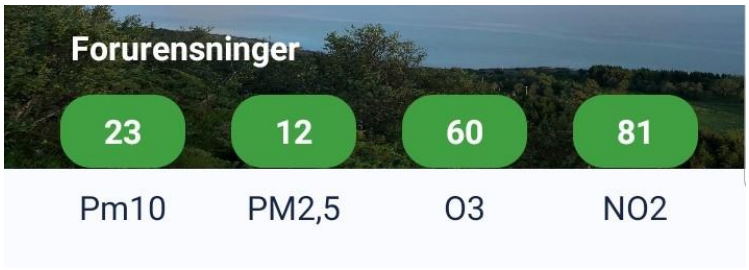
En viktig funksjonalitet for oss var at systemet skulle gi brukeren informasjon om luftkvaliteten fra stasjonen som befant seg nærmest brukeren. Informasjonen vi valgte å fremvise i systemet er den momentane luftkvaliteten, og stoffene som bidrar til forurensing. Dette valget tok vi utfra en spørreundersøkelse vi foretok før implementeringen. For brukerens del valgte vi også å fargekode disse slik at man lettere kan se forskjell på bra og dårlig luftkvalitet.

2.2.1 Momentane luftkvaliteten

En av tekstene som presenteres er den momentane luftkvaliteten representert ved *AQI*. Denne blir fremstilt i en sky øverst i venstre hjørne. Den viser hvor god eller hvor dårlig luftkvaliteten er der hvor brukeren befinner seg til enhver tid. Denne informasjonen gir brukeren muligheten til å se mer nøyaktig hvordan luftkvaliteten er i sitt nærområde, og ikke bare generelt for kommunen. Systemet oppdaterer luftkvaliteten på brukerens lokasjon hver time. I det systemet blir opprettet eller åpnet vil den vise fram den momentane luftkvaliteten i brukerens område. Om enhetens posisjon har endret seg mens systemet er oppe og kjører vil systemet automatisk oppdatere seg, og vise luftkvaliteten i det nye området. I spørreundersøkelsen som vi foretok under implementeringen av systemet svarte hele 49% av deltakere at de syntes det var veldig viktig å kunne se hvordan luftkvaliteten var i deres nærområde. Derfor ønsket vi at i det man starter systemet vil dette være det første brukeren blir møtt med, slik at man enkelt kan se hvordan luftkvaliteten er rundt dem i det øyeblikket.

2 Momentan luftkvalitet din posisjon

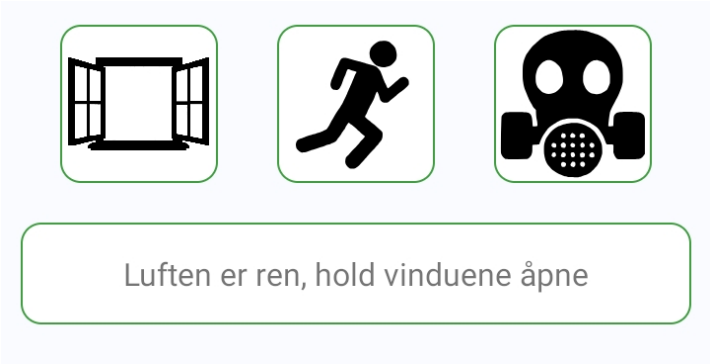
2.2.2 Systematisert forurensninger



3 Systematisert forurensing ved gitt posisjon

Brukeren for også informasjon om ulike forurensende stoffer det er i luften og mengden av disse på nåværende tidspunkt. Dette kan være spesielt viktig å vite for folk med Astma eller andre luftveissykdommer. I spørreundersøkelsen svarte 33% at det var veldig viktig å vise mengden av de ulike stoffene, noe som var en høy prosent av svar på "veldig viktig". Under skyen som viser den momentane luftkvaliteten ligger det derfor 4 fargede bokser. Disse viser hver sitt stoff og mengden av dette i *u/mg*[4]. Navnet på hvilket stoff det gjelder står under boksen. Hvorvidt luftkvaliteten er bra varierer fra stoffmengde til stoffmengde. Det kan være mye av ett stoff og lite av et annet, og man vil da se det både på verdi og bakgrunnsfarge. Systemet kan derfor vise dårlig momentan luftkvalitet for et stoff, selv om det likevel kan være trygt å for eksempel dra ut eller trene ute for de med luftveissykdommer.

2.2.3 Trykkbare ikoner med helsetips



4 Forklarende ikoner med tekstboks

Med utgangspunkt i spørreundersøkelsen så vi at hele 90% av deltagere valgte viktig eller veldig viktig med tips og helseråd utfra luftkvaliteten. Det var i tillegg 92% som ønsket informasjon om helserisikoen ved dårlig kvalitet på luften. Ettersom det var et såpass stort ønske om dette valgte vi å lage en implementasjon som var sentralt, og samtidig enkelt å forstå i systemet. Vi kom frem til at vi ønsket å oppgi informasjon om lufting i hjemmet, utendørstrening og om det var anbefalt med bruk av luftmaske eller ikke. Dette ble løst ved bruk av tre ikoner på siden med mest informasjon om luftkvaliteten på en lokasjon. Disse ikonene representerer hvert sitt formål og har en farget ytterkant som representerer hvor forsiktig man bør være. Ved trykk på ett av disse ikonene vil det også komme opp en

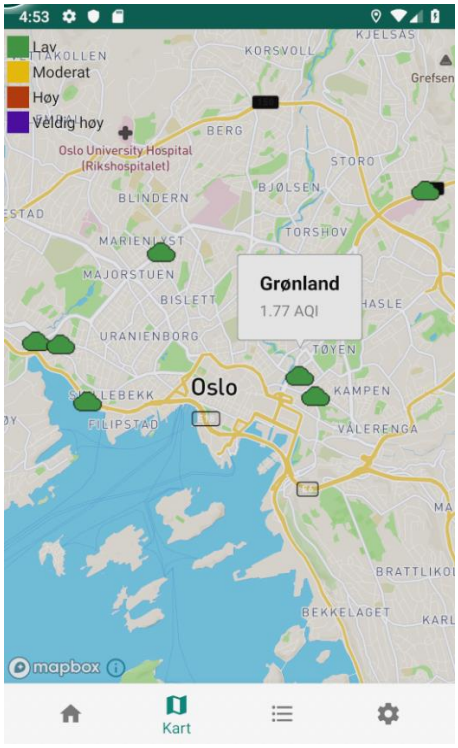
[4] u/mg = enheter pr milligram

tekstboks, med den samme løsningen for farget ytterkant. Inne i denne boksen vil det stå beskrevet en anbefaling til brukeren for bruk ved den nåværende luftkvaliteten.

Helt øverst i høyre hjørne er det også en knapp med et spørsmålstegn på. Ved trykk på denne knappen vil brukere få opp en skrollbar boks, som inneholder en del informasjon. Her er det beskrevet litt om hva hvert av stoffene som blir vist på siden er, samt en beskrivelse for *AQI* og fargekodene som blir brukt. I tillegg blir det beskrevet kort om helserisikoer ved for høyt inntak av de forskjellige stoffene, og det er beskrevet kort hva som er utsatt og hvilke sykdommer dette kan føre til.

2.3 Kart

I systemet valgte vi å fremstille stasjoner og luftkvalitet gjennom Mapbox sin *SDK*[5] for Android utvikling. Mapbox er en open source leverandør av digitale kart med høy tilpasningsevne, og henter sin data blant annet fra kilder som *OpenStreetMap* og *NASA*. Som nevnt i oppgavebeskrivelsen var det anbefalt å bruke et kart med et godt *WMS*[6]-klient-bibliotek fordi dette ville gjøre oppgaven lettere overkommelig. Etter å ha gjort litt søk angående dette kom vi frem til at valget sto mellom Google Maps og Mapbox. Og til slutt endte vi med å ta i bruk Mapbox. Dette er med tanke på at vi fikk inntrykk av at det fantes mye god dokumentasjon om hvordan biblioteket skulle og kunne tas i bruk. Det til tross for at Google Maps kanskje er et mer åpenbart valg på grunn av Googles velrenommerte merkevare i teknologi-bransjen.



5 Mapbox i bruk i systemet

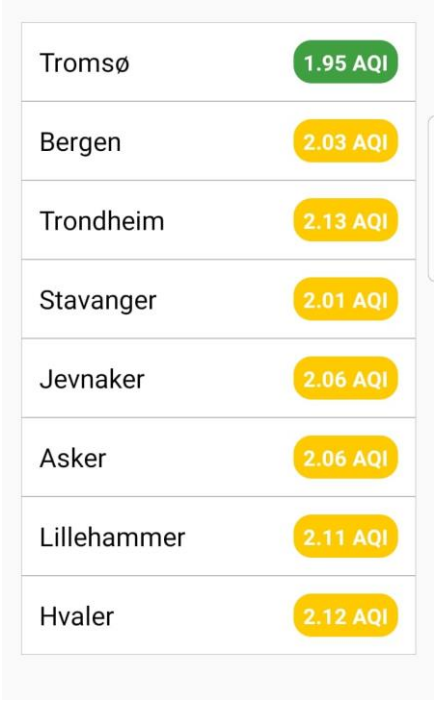
Det vi så på som spesielt viktig for kartets funksjonalitet i systemet var at det skulle være intuitivt og lett å tolke, og med det mener vi at det skulle være lett gjenkjennelig og at det skulle ta i bruk fargekoder som anses som universelt forstått. En av Mapbox sine store styrker

[5] SDK (software development Kit) = betegnelse for et sett av utviklerverktøy og koderammeverk
[6] WMS (web map service) = internasjonal standard for utveksling av kart over internett

var modifiserings- og tilpasningsmulighetene av kartet. Ideen var til å begynne med å bruke et sky-ikon som skulle fremstille luftkvaliteten, og at den skulle endre farge i samsvar med *AQI* nivå. Derfor valgte vi å erstatte en helt enkel markør, eller punkt, som vist i figur 5, med et egenprodusert sky-ikon. Disse kommer i fire forskjellige farger som er gjennomgående elementer i hele systemet slik at brukere får det lettere å forstå betydningen av dem. Ved å klikke på en av disse sky-ikonene vil mer informasjon om den valgte stasjonen komme i form av en hvit boks med tekst. Eksempel på dette har vi på figur 5 ved Grønland. Øverst i venstre hjørne valgte vi å inkludere et felt for å fremvise fargenes betydninger, dette til tross for det vi tidligere nevnte. I samme figur kan man se at skyene, i det øyeblikket skjermbildet ble tatt, var grønne, som vil si at luftforurensingen var lav ved de fremviste punktene.

2.4 Liste over kommuner i Norge

En funksjonalitet vi virkelig ønsket systemet skulle ha var at brukeren enkelt skulle kunne få presentert luftkvaliteten ikke bare på det stedet man er, men også andre steder i Norge. Dette ble delvis oppnådd ved bruk av punkter på kart som beskrevet over, ettersom det her blir vist en visuell representasjon av luftkvaliteten på utvalgte punkter. Men vi skjønte fort at denne funksjonaliteten vil fort bli tungvint ved "zooming" inn og ut på kartet. Derfor ønsket vi å vise dette på en annen og enklere måte.



6 Liste over kommuner med AQI nivå

Vi ønsket også at brukeren skulle ha muligheten til å tilpasse systemet etter egen preferanse, og dermed landet vi på at vi skulle implementere en liste over steder som enkelt viste luftkvalitet nivået (*AQI*) med en fargekode som indikerte hvordan luftkvaliteten var på dette tidspunktet. Denne listen skulle også ha muligheter for å endres av brukeren, ved at han/hun kan legge til, ta bort og endre på rekkefølgen i listen, slik at hver bruker selv kan bestemme hvordan listen skal se ut, dette vil vi komme tilbake til senere.

Som gruppe måtte vi også velge hva som skulle bli representert som steder i listen, og her hadde vi 4 valgmuligheter som gjorde det enkelt å hente ut den informasjonen vi trengte.

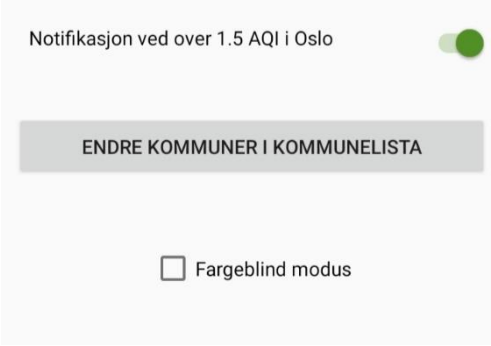
Valgene var: grunnkrets, fylke, kommune og delområde. Vi endte med å velge kommuner, ettersom vi følte dette tok for seg et område på en passe størrelse hvor det vil bli gitt en relevant representasjon for luftkvaliteten på et sted, men også ikke et område som er for lokalt, ettersom spørreundersøkelsen vår viste at det var et ønske at listen skulle gi et innblikk i forskjellen på luftkvalitet rundt om i landet.

For hver kommune i listen var vi ikke helt fornøyde med å kun vise luftkvalitet nivået for hver enkelt plass, men vi ønsket også at hvert element ikke skulle ta opp for mye plass eller vise mye informasjon, ettersom dette fort kan føre til et uoversiktlig design. Vi valgte derfor å gjøre hvert element i listen klikkbare, hvor man åpner en infoside som er veldig lik siden som viser informasjon om luftkvaliteten på min lokasjon. Denne siden inneholder en mer detaljert representasjon av luftkvaliteter, samt mer nøyaktig hvor målingen er funnet sted. Se figur 7. Her er innholdet nivåer for *AQI*, PM10, PM2.5, O3 og NO2, samt tre bokser som ved tykk av brukeren viser en tekstboks med helseanbefalinger for luftkvaliteten på dette stedet.

7 Ved trykk på element i liste

2.5 Innstillinger

En viktig funksjonalitet for oss var at brukere selv skulle få friheten til å personalisere systemet sitt. Det er derfor vi opprettet tre mindre funksjonaliteter i Innstillingene.

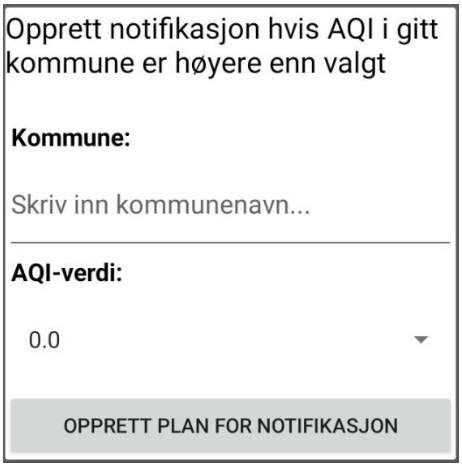


8 De tre hovedfunksjonene i Innstillinger

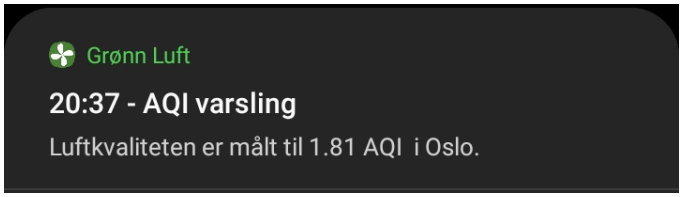
2.5.1 Varsling

Den første funksjonaliteten gir brukere muligheten til å opprette en egendefinert notifikasjon. Den er laget slik at den ikke slår ut før verdien før luftkvalitet i en kommune overstiger en selvvalgt verdi. Systemet vil ved opprettelse av notifikasjon jobbe i bakgrunnen, og «plinge» dersom betingelsene skal være møtt. Da vil også en enkel melding som viser tidspunktet luftkvaliteten ble målt, samt *AQI*-verdien som ble målt. En notifikasjon vil på det meste komme hver 6. time, slik at den ikke oppfattes som plagsom, men likevel kan virke informativ og nyttig.

Notifikasjon er viktig for brukere som ønsker å vite om for eksempel hjemkommunens luftkvalitet potensielt er skadelig å puste inn. I spørreundersøkelsen mente hele 88% at det å kunne sette varsling etter eget behov enten var viktig eller veldig viktig. Derfor har vi forsøkt å gjøre varselsystemet så enkelt som mulig, slik at flest mulig brukere av systemet skal kunne benytte seg av dets fulle potensiale.



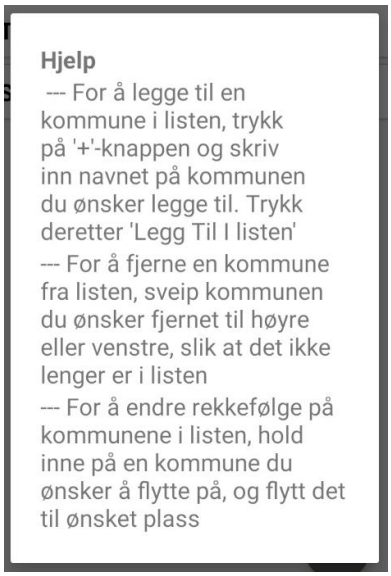
10 Brukergrensesnitt for å opprette plan om notifikasjon



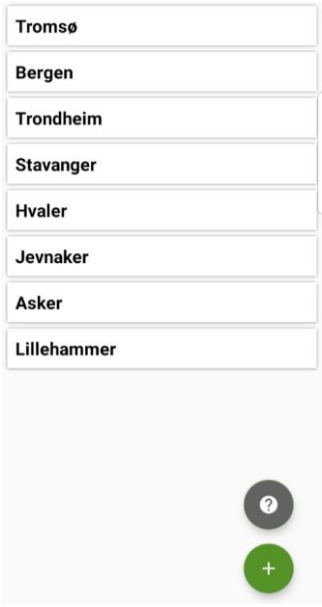
9 Eksempel på en varsling

2.5.2 Endre kommuner i kommunelista

Denne funksjonen er tilknyttet listefunksjonaliteten nevnt tidligere. Den brukes for å legge til, fjerne og sortere kommunene i listen. Hvis man planlegger å ha en del kommuner i lista, for eksempel til loggføring ved forskning, kan dette vise seg å være en nyttig funksjon. På samme måte som man kan velge kommune når man skal opprette varsel, kan man også her velge kommunene man ønsker å fokusere på. I tillegg til dette, finnes det også en liten guide som forklarer hvilke minimale gester som skal til for å fjerne, legge til og sortere elementene i lista dersom man skulle være usikker.



11 Hjelpedialog som forklarer hvordan redigere lista



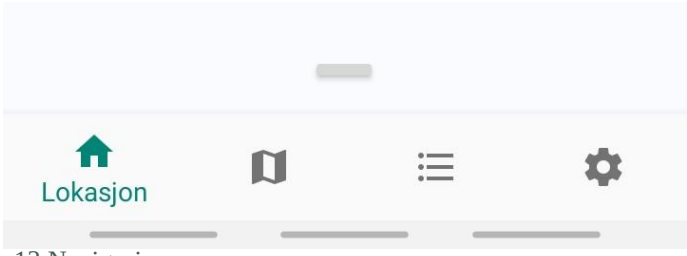
12 Kommuner som skal vises i lista og to knapper for informasjon om redigering og å legge til et element

2.5.3 Fargeblindmodus

Den tredje og siste funksjonen endrer alle fargene som er med på å beskrive luftkvaliteten. Denne modusen er hovedsakelig for folk som har problemer med å skille mellom enkelte farger, men kan like gjerne brukes av folk som er misfornøyd med fargene vi alt hadde valgt ut.

2.6 Navigeringsmeny

Med tanke på alle funksjonalitetene i systemet, er det viktig at brukeren lett skal kunne navigere seg rundt i systemet vi har laget. Det er derfor vi opprettet en liten meny helt nederst i applikasjonen.



13 Navigasjonsmeny

Denne funksjonalitet lar bruker navigere seg mellom hver funksjonalitet uten å måtte trykke tilbake på enheten systemet blir kjørt på. Denne navigasjonsmeny består av 4 forskjellige symboler. Hvert av disse symbolene fungerer som en knapp med hver sin funksjonalitet. Om en av de blir valgt vil brukeren bli videreført til den tilsvarende funksjonaliteten. Symbolene er universelt forstått slik at det ikke er noe tvil om hvilke funksjonaliteter man ber systemet kjøre. Systemet vil ved oppstart starte på «hjem»-siden. Hjem-siden vil vise brukeren

luftkvaliteten i deres nærområde. Meny 2 er representert med et kartsymbol som vil sende brukeren over til systemets kartfunksjon. Meny 3 er representert med et listesymbol. Denne vil sende brukeren over til liste over utvalgte kommuner i en liste. Tilslutt meny 4 blir representert med et tannhjul. Dette tannhjulet vil sende brukeren over til innstillingene til systemet. For at skal lett forstå hvilken funksjonalitet man befinner seg på vil symbolet man trykker på i systemet bli forstørret og bli markert med en farge. Det vil også da dukke opp en tekst under symbolet som forklarer med tekst hvilken funksjonalitet systemet viser. Se figur 13.

3 Kravspesifikasjon og modellering

I kravspesifikasjonen skal vi beskrive de viktigste funksjonelle og ikke-funksjonelle kravene til systemet. Noen av funksjonalitetene er ikke inkludert i systemet grunnet forskjellige årsaker nevnt senere i prosessdokumentasjonen, dette er funksjonaliteter vi ville utviklet ved oppdateringer av systemet i fremtiden. Alle kravene er beskrevet i en liste under, de som står i kursivt er de som ikke er inkludert i systemet.

3.1 Kravspesifikasjon

3.1.1 Funksjonelle krav

Systemet skal ha en kartfunksjon som viser luftkvaliteten rundt om i landet
Systemet skal kunne måle momentan luftkvalitet på en gitt posisjon
Systemet bør rangere luftkvaliteten på en gitt posisjon
Systemet skal kunne vise hvordan luftkvaliteten har utviklet seg de siste 24 timene
Systemet skal kunne kalkulere forventet luftkvalitet de kommende dagene gitt værvarslingen
Systemet skal vise en egendefinert liste over luftkvalitet i kommuner
Systemet bør kunne systematisere luftkvalitet etter stoff
Systemet bør ha mulighet for push-varsling
Systemet bør inneholde en fakta side som har informasjon om forskjellige stoffer
Systemet bør beskrive luftkvalitetens generelle påvirkning i samfunnet (Tiltak, Helseeffekter, Luftforurensing, regelverk)
Systemet skal inneholde en navigasjonsmeny

3.1.2 Ikke-funksjonelle krav:

Systemet må utnytte et *API* for å hente korrekt data

Systemet må vise korrekte data, både om luftkvalitet og posisjoner i kartet

Systemet må kunne hente brukerens posisjon

Systemet må kunne lagre informasjon i minst 24 timer

Systemet må skrives på norsk, og ha lesbar skriftstørrelse

Systemet må være lett å forstå for enhver som bruker den

Systemet må kjøres raskt nok til at det ikke blir irriterende

Systemet må være lett å navigere

Systemet må ha tilnærmet null feil ved brukerinteraksjon

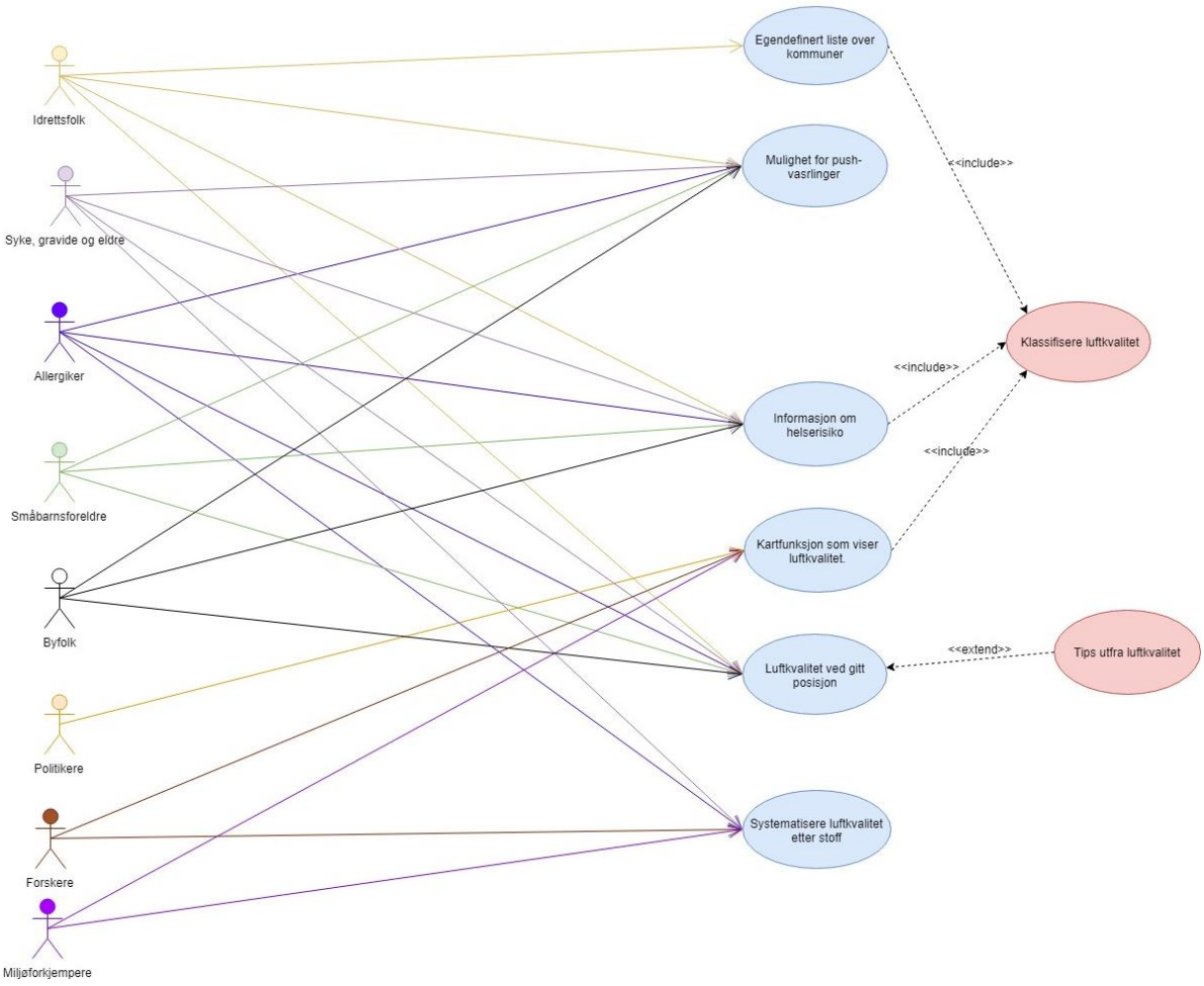
Systemet må kunne jobbe i bakgrunnen på smarttelefonen

I utviklingen har vi fokusert på å holde systemet velfungerende og godt strukturert. Vi delte alle aktivitetene inn i fragments som hver for seg fokuserte på sin funksjonalitet. Alle klassene gjorde sin oppgave, og ikke noe mer enn det. Slik oppnådde vi høy *kohesjon*[7]. Den høye *kohesjon*en gjorde at vi lett kunne endre kode dersom feil skulle oppstå. Potensielle feil ble også lettere å identifisere. Objektene i systemet vårt har av samme grunn lav *kobling*[8]. Det vil si at de som regel er avhengig av få andre objekter for å utføre sin oppgave. Vi tok inspirasjon fra et *Model View Presenter*[9]-mønster, ved å ha en MainActivity som hadde muligheten til å bytte mellom de fire Fragmentene som hver utførte sin funksjon i et View. Dette er standarden for utvikling av android-apper, men vi valgte likevel ikke å følge mønsteret slavisk. Vi forenklet den litt for å passe både våre ambisjoner med systemet og gruppas erfaringer med Android Studio. All koden er skrevet på engelsk, slik at den har potensiale til å forstås av personer med generell programmeringsbakgrunn, men likevel ikke kan norsk. Vi har også kommentert deler av koden for å gjøre den lettforståelig for utviklere.

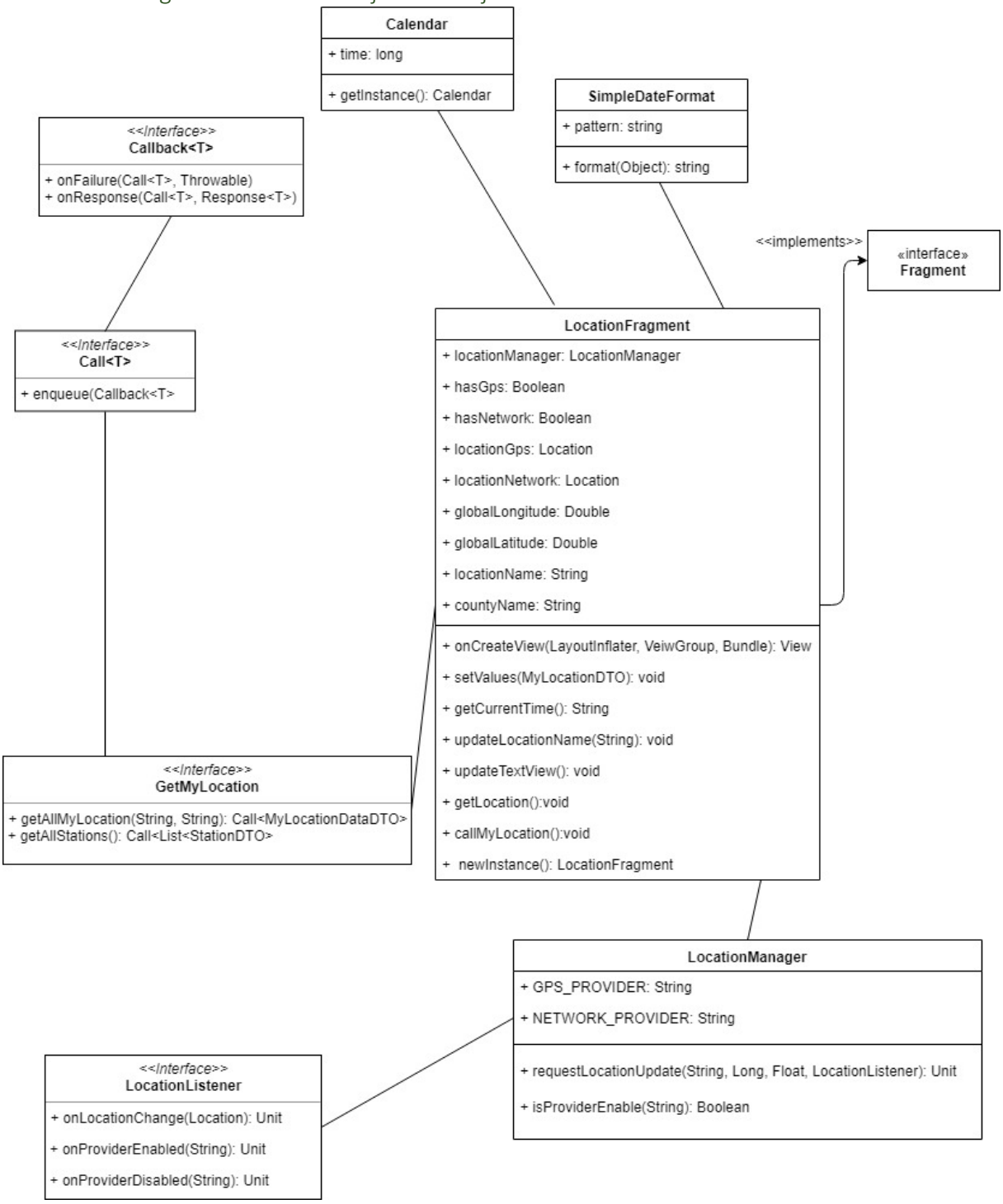
[7] Kohesjon = Et mål på hva slags ansvar et objekt har og hvor fokusert ansvaret er
[8] Kobling = Et mål på hvor sterkt et objekt er knyttet til andre objekter
[9] Model View Presenter = en mal for hvordan aktivitetene knyttes sammen i en Android-app

3.2 Modeller

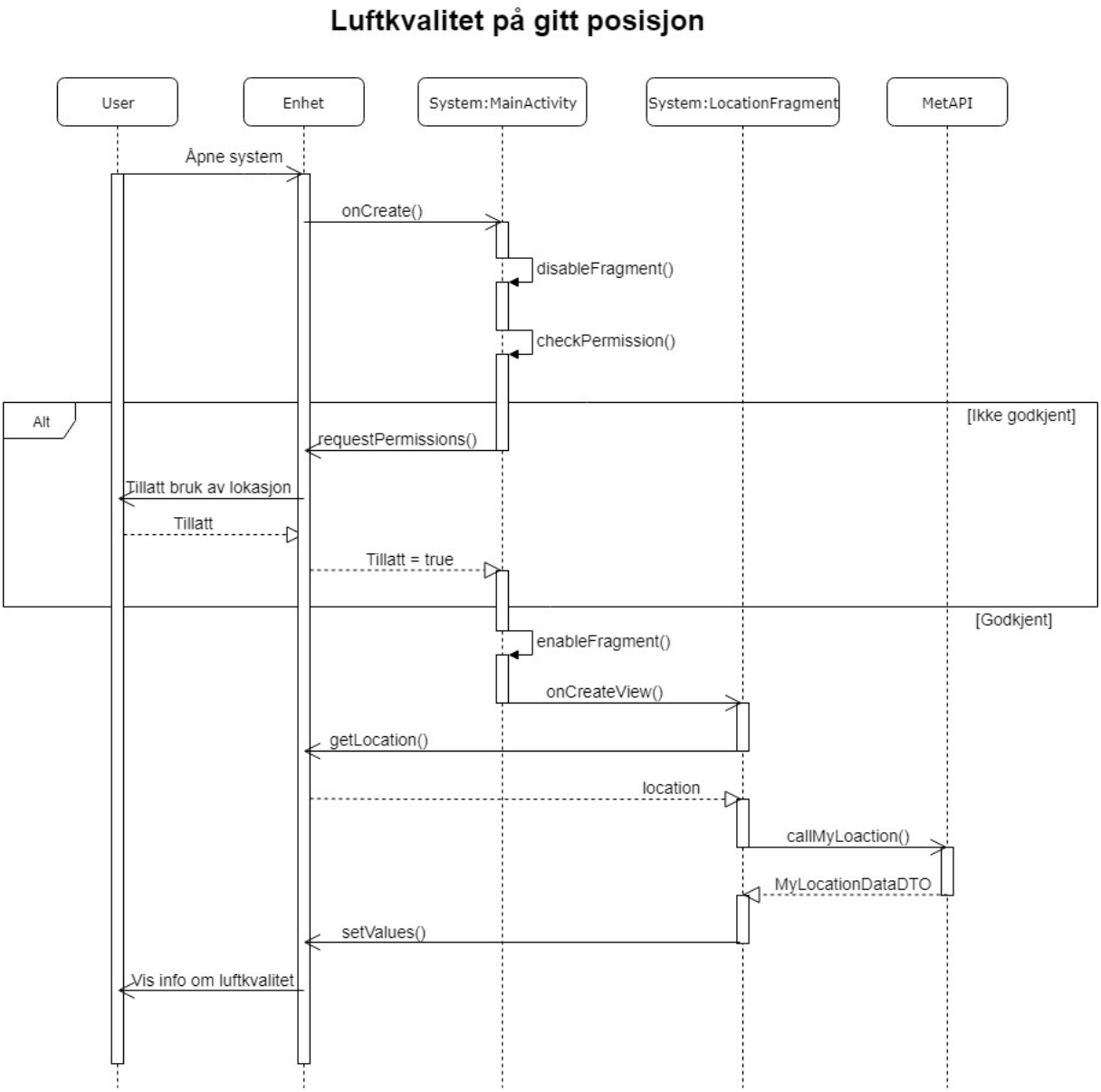
3.2.1 Use-case modell



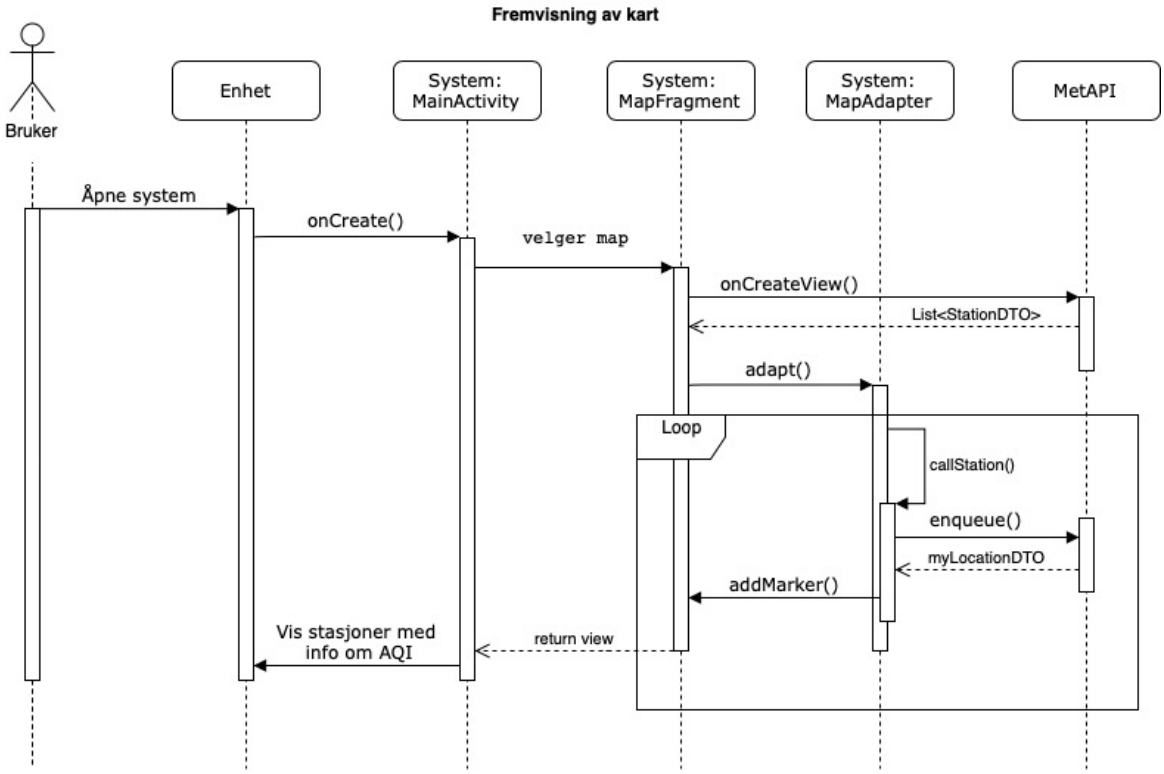
3.2.2 Klassediagram for «Min lokasjon» funksjonalitet



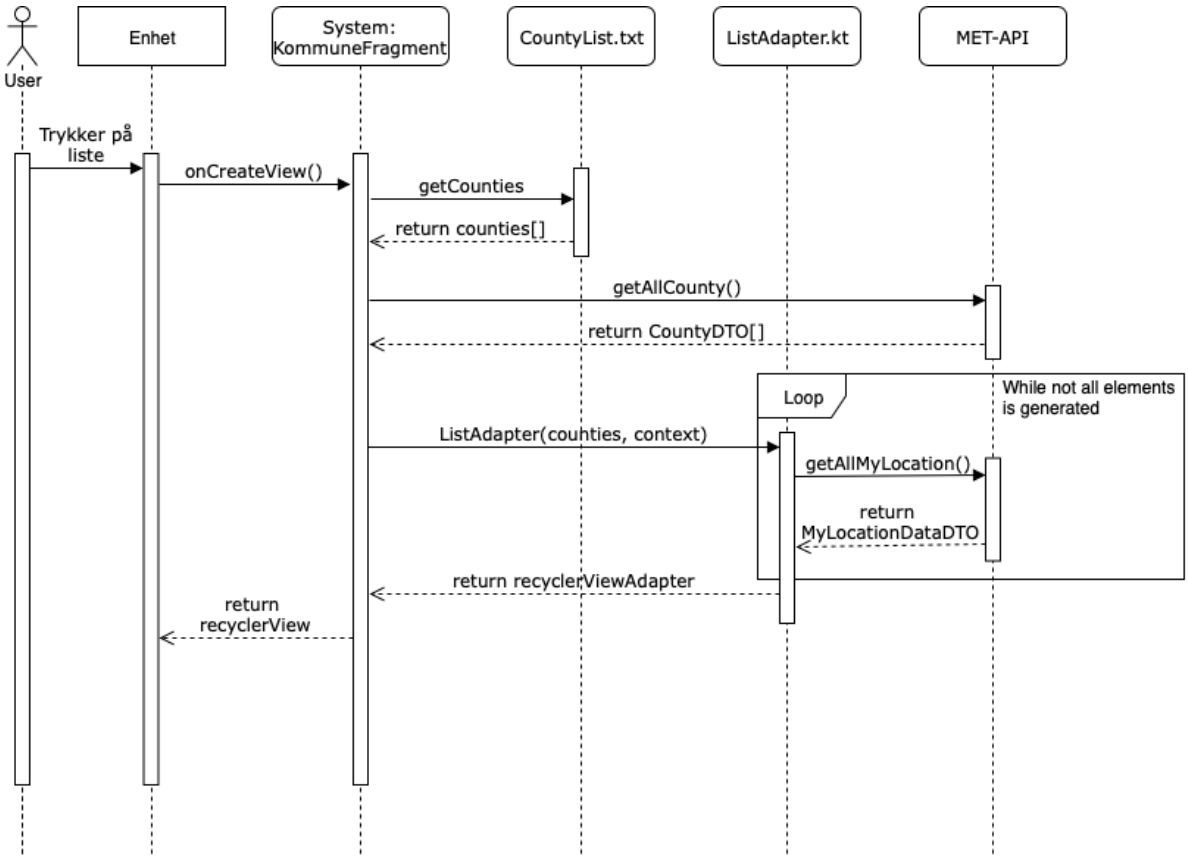
3.2.3 Sekvensdiagram for luftkvalitet på gitt posisjon



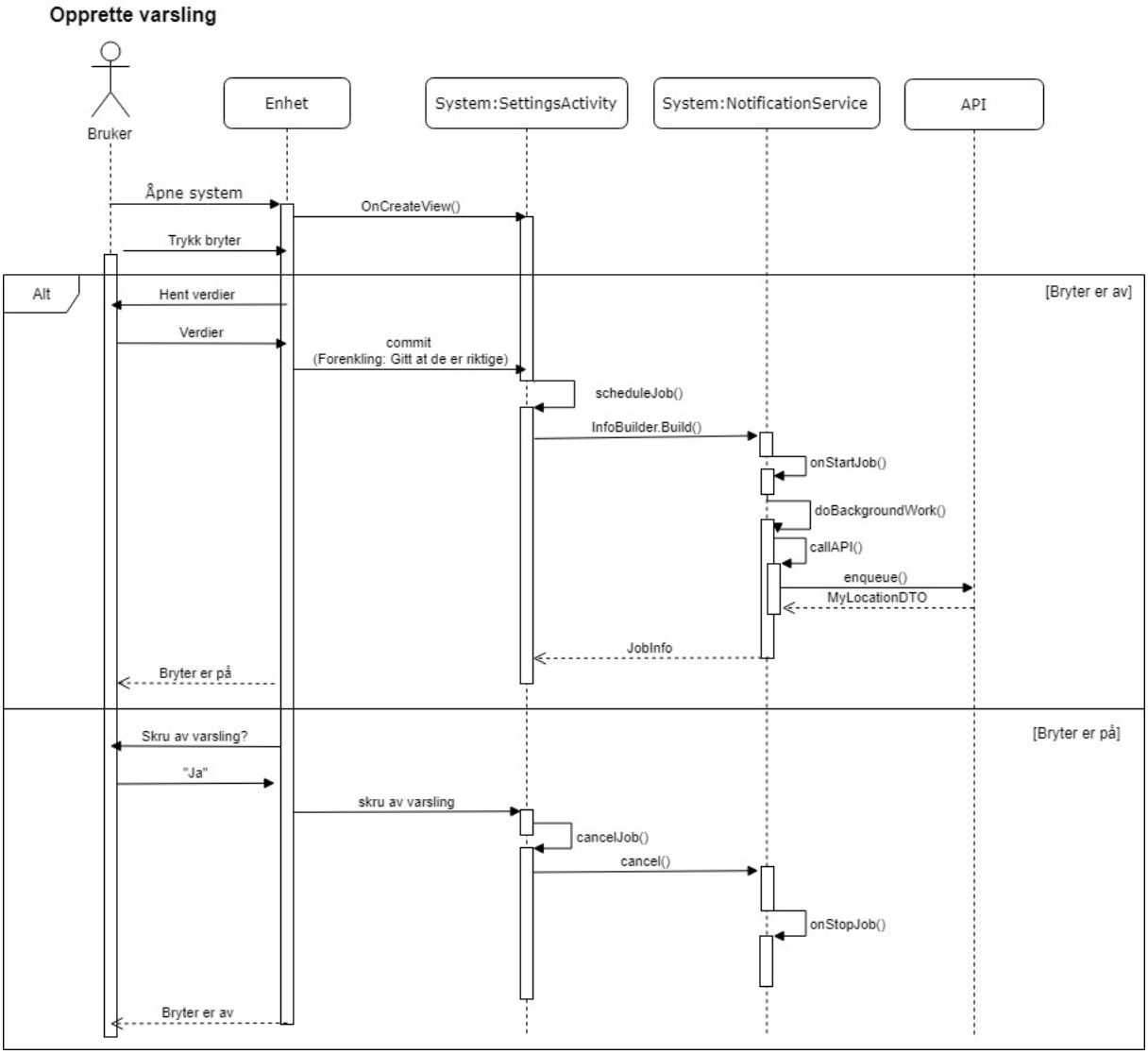
3.2.4 Sekvensdiagram for fremvisning av kart



3.2.5 Sekvensdiagram for liste over kommune



3.2.6 Sekvensdiagram for opprettelse av varsling



4 Produktdokumentasjon

4.1 Utnyttelse av *API*

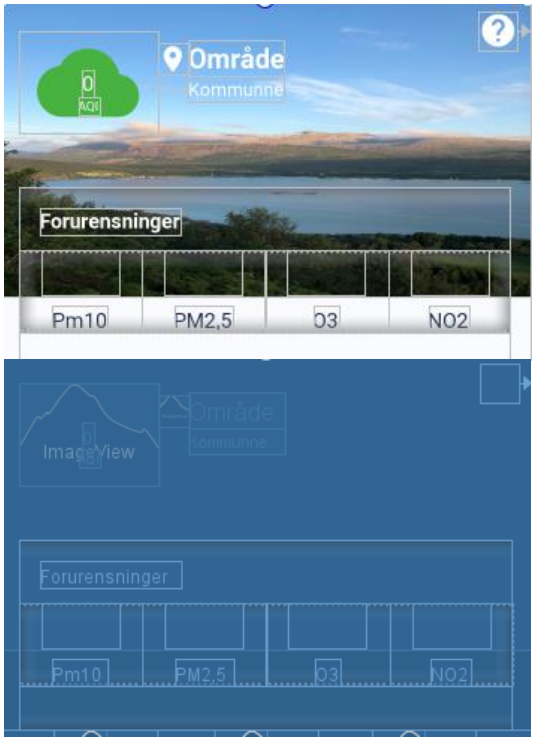
I starten av systemutviklingen, begynte vi å jobbe rundt *API*-et[10] og implementasjonen av den. Vi utnyttet *Retrofit*[11], som er en klasse som enkelt kan konvertere et *JSON*[12] tekst fra et *API* til et *Kotlin*[13] (el. Java)-grensesnitt. I *RetrofitClientInstance.kt* opprettet vi en basisinstans av *Retrofit*, som tar inn url-en man skal gjøre de forskjellige kallene på. Deretter oppretter vi grensesnitt (eks. GetCalls.kt) som tar inn resten av kallet til url-ene. Ettersom vi valgte å løse *API*-et på denne måten var vi også nødt til å lage *dataklasser*[14] for de ulike delene av *API*-et vi hentet ut. Alle dataklassene ligger i en egen mappe (DTO), og har «DTO» i slutten av filnavnet. Vi fant et nyttig verktøy som tok inn ren *JSON*-tekst, formaterte den og opprettet *dataklasser* basert på informasjonen den hentet ut. Dette er en plugin som heter *JSON to Kotlin Class* og kan lastes ned i biblioteket til Android Studio. Det gjorde det lettere å holde styr på *API*-kall som inneholdt mange forskjellige variabler, og systematiseringen av disse. Til slutt fullførte vi dannelsen av *Retrofit* ved å kalle på den. Det er i tillegg viktig at enheten er koblet til internett, noe som kan sikres ved å legge til «<uses-permission android:name="android.permission.INTERNET"/>» i manifest-filen. Det er anbefalt at man har to funksjoner som hver tar for seg hendelsesforløpene dersom tilkobling til *API*-et er en suksess og dersom det mislyktes. Disse kalles henholdsvis *onResponse* og *onFailure*. *onFailure* holder seg tilnærmet lik hver gang den brukes i programmet. Den vil vise en *Toast*[15] som forklarer brukeren at det er noe feil med tilkoblingen, og oppfordrer til å prøve igjen senere. Metoden *onResponse* vil være unik for hver funksjonalitet. Metoden returnerer dessuten alltid en dataklasse med alle relevante verdier hentet fra *API*-et. Metoden vil derfor forklares nærmere de stedene den er tatt i bruk. Det er dessuten viktig å huske at alle *onResponse*-metoder kjører asynkront på en annen tråd. Man må derfor bli ferdig med alle operasjoner på *API*-et inne i metoden. Dette kallet ble brukt i nesten alle delene av programmet vårt. Vi ble derfor svært avhengige av at det var implementert riktig fra starten av. For å ta i bruk alle disse funksjonene, måtte vi implementere

[10] API (Application Programming Interface) = En måte å kommunisere med Meterologisk institutt, og hente deres målinger
[11] Retrofit = En klasse brukt for å konvertere JSON tekst fra API til Kotlinklasser
[12] JSON = formatet informasjonen hentet ble presentert på
[13] Kotlin = Programmeringsspråket vi hovedsaklig utnyttet i systemet
[14] Dataklasse = klasse som inneholder data
[15] Toast = en midlertidig melding som vises på enheten

'com.squareup.retrofit2:retrofit:2.5.0', 'com.squareup.retrofit2:converter-gson:2.5.0' og 'com.squareup.retrofit2:adapter-rxjava2:2.5.0' i gradle-filen.

4.2 Min lokasjon

For å vise luftkvalitet og ulik informasjon fra plassering til enheten eller en hvilken som helst kommune, blir det brukt samme design bare med noen små forskjeller. Det er to filer: *activity_countylocation.xml* og *fragment_location.xml*. Hele designet er bygget med *ConstraintLayout*. I layoutet er det brukt *imageviews*, *view*, *textview* og andre layouts for å vise ulik informasjon og bilder. Øverst er det et *imageview* som viser en drawble av en sky, som har en *textview* over seg som forteller hva *AQI* nivået er. Ved siden av er det nye *imageview* med *textview* rett ved siden av uten mellomrom. *Textview* sier noe om hvilket område som vises, og under er det et nytt *textview* som sier hvilken kommune det er. Helt øverst i høyre hjørnet befinner det seg *textview* som ved click viser bruker informasjon om *AQI* og de ulike stoffene. Under dette igjen er det en vertikal *LinearyLayout*. Den viser en *textview* med overskriften «Forurensinger». Videre under er det en ny horisontal *LinearyLayout*. Inni den igjen er det 4 nye horisontale *LinearyLayout*. Disse viser to *textviews*. Den øverste er formet som en boks, og forteller noe om mengden det er av et stoff. *Textviewet* under sier noe om hvilket stoff det er.



14 Min lokasjon informasjon sett fra design perspektiv

En av systemets funksjonaliteter er at den henter enhetens lokasjon og viser luftkvaliteten i dette område. For at systemet skal kunne gjøre dette er det nødvendig at den vet koordinatene til enheten, men før systemet kan gjøre dette er det en del ting som må på plass først. Aller først må vi gi systemet tilgang til nett, og muligheten til å hente lokasjon fra enhet. Dette gjøres i systemets *Manifest*. For å få tilgang til lokasjon spør systemet om to forskjellige tilganger. En er for presise lokasjoner «android.permission.ACCESS_FINE_LOCATION»,

og den andre er for den tilnærmede lokasjonen «android.permission.ACCESS_COARSE_LOCATION». Etter å ha gitt systemet tilgang til å få lokasjon fra enhet er det også viktig at vi spør om tillatelse fra brukeren av systemet. Dette gjør systemet i *onCreate* i *MainActivity.kt*. Først er det viktig at funksjonaliteten som krever enhetens lokasjon ikke er aktivert. Derfor vil den alltid som standard være deaktivert. Systemet sjekker så om alle tilganger er gitt eller ikke ved hjelp av funksjonen checkPermissons(permissionArray: Array<String>). Denne tar en array med tilganger og returnerer tilbake en boolean-verdi som tilsvarer hvorvidt alle tilgangene i arrayen er gitt eller ikke. Om alle tilganger er gitt vil funksjonaliteten bli aktivert. Derimot om de ikke er gitt, vil systemet spørre bruker om disse tilgangene. Dette gjøres ved å kalle på *requestPermissions*. Denne spør brukeren om hvilke tilganger den vil tillate og ikke. Etter hvert kall på metoden *requestPermissions* vil en metode kalt *onRequestPermissionsResult* alltid kjøre og ta imot resultatet. Om tilganger ikke ble gitt vil den sende ut en *Toast* til brukeren med en beskjed om dette. Hvis alle tilganger ble gitt vil metoden aktivere funksjonaliteten som trenger enhetens lokasjon. I tillegg til dette, vil systemet ved oppstart lage en kanal for notifikasjoner. Hensikten med denne er nærmere forklart i produktdokumentasjonen til Innstillinger.

Når systemet har fått tilgang til lokasjonen kan den endelig gå videre med funksjonaliteten som henter lokasjonen og koordinatene til enheten. Dette gjøres ved at fragmentet *LocationFragment.kt* blir kjørt og vist på siden. Ved oppstart av *LocationFragment.kt* vil selve viewet bli satt opp, men så må vi få noen verdier inn i det. Dette gjøres ved at den senere kaller på funksjonen *getLocation()*. I denne metoden brukes det *LocationManager*[16]. *LocationManager* er en klasse i android location biblioteket. Den gir tilgang enhetens lokasjon tjenester. Disse tjenestene gir systemet muligheten til å motta periodisk oppdateringer av enhetens geografiske plassering. Ved hjelp av disse bruker vi metoden *isProviderEnabled* i *LocationManager* til å sjekke om enheten bruker GPS eller nettverket for å hente den geografiske plasseringen. Systemet vil så hente enhetens plassering ved hjelp av GPS og nettverk utafra hvem som er tilgjengelig eller ikke. Dette gjøres ved metoden *requestLocationUpdates* i *LocationManager*. For at denne metoden skal kjøre trenger den noen parametere. Først hvilken leverandør den skal sjekke, GPS eller nettverk. Hvor ofte skal den lokasjonen oppdateres, 1000 er lik 1 sekund. Hvor mye på plassering endres for at den skal oppdateres, 1000 er lik 1km. Og tilslutt en listener som heter *LocationListener*[17].

[16] LocationManager = klasse som gir tilgang til systemets lokasjons tjenester
[17] LocationListener = brukes til å motta varsler fra LocationManager når plasseringen er endret

LocationListener er en interface i det samme biblioteket. Denne lytter og oppdager når plasseringen har blitt endret. Den har 3 metoder som man må implementere:

- *onLocationChanged*, kalles når lokasjon blir endret
- *onProviderDisabled*, kalles når leverandør er deaktivert av bruker
- *onProviderEnabled*, kalles når leverandør er aktivert av bruker

For oss er det bare interessant å bruke *onLocationChanged*. Så lenge den nye lokasjonen ikke er lik null, vil den hente plasseringens *latitude*[18] og *longitude*[19], og lagre disse til globale variabler som blir brukt senere.

Til slutt når systemet har hentet koordinatene den trenger kan vi hente ut alle verdiene vi trenger. Dette gjør systemet ved å kalle på metoden *callMyLocation*(). Denne funksjonen gjør et *Retrofit* kall på *API*et «https://in2000-*API*proxy.ifi.uio.no». Men for å kunne hente informasjon om luftkvalitet gitt en lokasjon, må vi modifisere det kallet litt, og sende med variabler som *latitude* og *longitude*. Dette gjør vi ved hjelp av *@Query*. Den vil bygge en streng for oss som gjør at kallet mot *API*et blir slik: «?lat=x&lon=y». Hvor x er *latitude* og y er *longitude*.

Fra dette kallet henter vi inn all informasjon som vi trenger og skal sette inn i fragementet. Dette gjøres i kallets *onResponse*. Verdiene blir hentet fra *dataklassen MyLocationDataDTO*. Først henter vi navnet på stedet det blir målt, og deklarerer den til en global verdi *locationName*. Så henter vi inn navnet på byen og deklarer den til variabelen *county*. Etter dette skal vi sette inn alle de nødvendige verdiene i alle viewene i fragmentet. Dette gjøres ved hjelp av metoden *setValues(MyLocationDataDTO)* som vi sender med data klassen. I *setValues()* henter vi frem alle de nødvendige viewene som vi skal sette inn verdier for. Men dataklassen har en liste med flere tidspunkter som igjen inneholder all informasjonen som vi trenger. Vi er må derfor hente fra klokkeslettet på enheten slik at vi vet hvilken informasjon vi skal hente frem til hvilken tid. Dette gjøres ved å kalle på metoden *getCurrentTime()*.

[18] Latitude = Geografisk koordinat som spesifiserer nord-sør posisjon på jordas overflate
[19] Longitude = Geografisk koordinat som spesifiserer øst-vest posisjon på jordas overflate

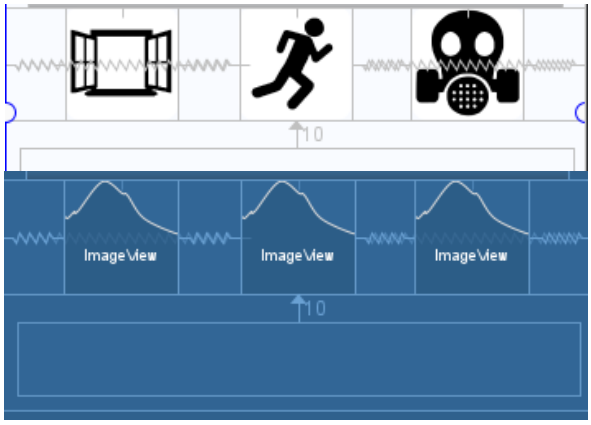
Metoden *getCurrentTime()* tar i bruke klassen *Calendar*[20] fra andoird biblioteket. Med denne klassen kan vi hente den nåværende tiden og dato ved å kalle *Calendar.getInstance(),* og deklarer den til variablen *calender*. Men for at tidspunktet vi henter skal ha likt format slik som det står i data klassen vår. Bruker vi *SimpleDateFormat*[21] som er enda en klasse i android biblioteket. Vi lager så et streng format «yyyy-MM-dd'T'HH':00:00Z'», og deklarer den til variabelen *timeFormat*. Så kan vi da bruke den offentlige metoden format fra klassen *SimpleDateFormat* på *timeFormat*. Men format trenger datoen som parameter, så vi sender derfor med variabelen time fra *Calendar* klassen og får: *timeFormat.format(calendar.time).* Resultatet av dette vil bli returnert av *getCurrentTime(),* og deklarert til variabelen *stringTime* som befinner seg inni *setValues()* metoden vår.

Når vi har funnet tiden kan vi gå igjennom hele listen med tider fra data klassen ved hjelp av en for loop. Så om en av elementene i listen har samme dato og tid som enheten går vi videre og setter verdier. Siden alle verdiene vi henter fra dataklassen er av type Double vil vi at det skal være av type Integer grunnet senere bruk. Vi bruker derfor metoden *roundToInt()* på hver verdi. Hver verdi vil så bli brukt for å kunne sette fargekoder på skyen (som inneholder den momentane luftkvalitet) og boksene (som inneholder mengden av stoffene). Ved hjelp av utrykket *when*, og tabellen vist i brukerdokumentasjonen endrer vi drawable'en til *imageviewet* med skyen, og bakgrunnsfargen til *textviewene* med mengden til hvert stoff. Tilslutt setter vi verdiene inn i *viewet* sin tekst, men verdiene må være en streng. Vi bruker derfor metoden *toString()* på verdiene før de blir satt inn. Vi har valgt å lage fire forskjellige skyer fra som ligger drawable mappen som representerer luftkvaliteten ved fargekode:

- *ic_cloud_low.xml*
- *ic_cloud_moderate.xml*
- *ic_cloud_high.xml*
- *ic_cloud_veryhigh.xml*

[20] Calendar = Klasse som lar deg hente dato og tid på enheten
[21] SimpleDateFormat = Konkret klasse for formatering og analyse av datoer på en lokalfølsom måte

Når det kommer til de trykkbare ikonene er det laget tre ikonene som *textViews*, hvor det er satt en egne produsert bakgrunner, som man finner i *drawable* mappen. Det er selvforklarende hvor brukeren skal trykke, ettersom det lagt inn en kant rundt bilde. Dette former en ramme som også har en farge som visualiserer om det er risikabelt eller ikke. Hver boks har ikoner som er entydige så enhver bruker forstår i grove trekk hva det omhandler. Det er i tillegg finner vi et *textView* under de tre bildene. Denne kommer til syne ved trykk på et ikon, og det blir fremstilt tekst som utrykker noen anbefalinger til brukere. Dette *textViewet* har også en kant, hvor fargen tilsvarer fargen til kanten det ikonet som ble trykket på har.



15 Trykkbare ikoner sett fra design perspektiv

De trykkbare bildene blir programmert i *setValues()* metoden, og blir brukt på samme måte både i filene *LocationFragment.kt* og *CountyLocationActivity.kt*. De tre bildene som er definert i activity_location.xml filen blir lagt til som verdier i koden, og deretter blir hvert bilde lagt til et array. Så blir det verdiene for både fargen på rammen rundt bilde og strengene som skal settes inn i tekstboksen blir initialisert. Etter vi har funnet den nåværende tiden vil vi deretter sjekke nivået på luftkvaliteten og sette verdier til teksten og fargen på kantene til de tre klikkbare bildene. Dette blir gjort ved at vi finner nåværende *AQI* og ut ifra denne verdien setter *BackgroundResource* til xml filen som inneholder bakgrunnene og har rett farge. Vi har her valgt å lage fire forskjellige filer fra drawable som representerer luftkvaliteten ved fargekode:

- *Textviewshadow_low, grønn.*
- *Textviewshadow_moderate, orange.*
- *Textviewshadow_high, rød.*
- *Textviewshadow_veryhigh, lilla.*

Når alle verdiene er satt opprettes det en *onClickListener* for hvert bilde litt lenger ned i koden. Nærmere bestemt etter vi har satt verdier for alle stoffene og lagt disse til i designet. I *onClickListener* sier vi at hvis bilde blir tykket på av brukeren, vil teksten i tekstboksen vises som den teksten som ble definert for den luftkvaliteten stedet har akkurat nå, og kantene til

den samme fargen som kantene til bildet har. Det er også en *onClickLsitener* for spørsmålstegn knappen i høyre hjørne. Ved trykk på denne gir det brukeren en visning av *info_substancees_poppop.xml*.

4.3 Kart

Mapbox inkluderer et fullverdig bibliotek og de har på deres egne nettside god dokumentasjon over hvordan det kan bli tatt i bruk. Biblioteket ble enkelt implementert inn i systemet gjennom *Android Studios* sin *build.gradle* ved å legge til en kodelinje, som henter inn den nyeste versjonen av Mapbox, under *dependencies*. Kort fortalt kan vi se på *gradle* som et utviklingsverktøy som simplifiserer implementasjonsprosessen av forskjellige vanlige biblioteker, deriblant *Retrofit* og *Mapbox* som vi begge har brukt i vårt system.
Rent teknisk og kodemessig består kart-funksjonaliteten av en enkel *onCreateView()* som returnerer et view. Den er til fordi vi i stedet for å ta i bruk det typiske *AppCompatActivity()*for hvert av funksjonalitetene, danner klasser av type *fragments()*. Det medfølger at vi kan kunne ta i bruk navigasjonsmenyen som vist i figur 13. I *activity_map.xml* finner man et *ConstraintLayout* som videre inneholder en visning av *Mapbox*. Sammen med viewet som returneres fra *onCreateView()* og oppsettet i xml-filen får vi visningen slik den er i systemet. Her har vi også valgt å implementere indikator som forteller noe om sky-ikonene og betydningen av disse. Dette ble gjort ved å neste *LinearLayouts* med to *TextView*, der fargekode og betydning fremstilles.

Mapbox i seg selv er gratis å ta i bruk, men om tilfellet skulle bli slik at systemet bestiger et antall på minst 50 000 brukere vil det medfølge en kostnadssum alt ettersom hvor høy trafikken systemet vil inneha. Av den grunn krevdes det at vi hentet en instans tilhørende en spesifikk bruker registrert hos Mapbox.

Koden, utenom det nevnte, består i hovedsak av life cycle-metoder. Disse er enkeltfunksjoner som forteller hvordan systemet vil handle ved visse spesialtilfeller, blant annet ved oppstart av systemet, om den skulle pauses eller om den skal avsluttes (*onStart, onPause, onCreate*). Alle disse står oppgitt med god dokumentasjon på Mapbox sine nettsider. Slik systemet er nå har vi ikke gjort noen merkverdige endringer på akkurat disse. Et eksempel for hvordan disse eventuelt kunne bli tatt i bruk under videreutvikling er ved å implementere *onClick()*-funksjoner. Med å for eksempel lage en funksjon som gjør slik at en ny markør skal dukke

opp ved et vilkårlig klikk på kartet. På grunn av at kartet lastes opp ved *onCreate(),* som er da kartet initialiseres, vil vi derfor trenge en *onResume*()-funksjon som sammen med en annen life cycle-funksjon tillater at endringer ved kartet gjøres selv etter at den allerede har blitt opprettet.

Vår egentilpasning av Mapbox ligger, som nevnt i brukerdokumentasjonen, ved bruk av egenspesifiserte markører for hvert av stasjonspunktene. En gjennomgående ting i dette prosjektet har vært måten vi henter inn data fra *API*-et. Dette har blitt gjort ved bruk av *Retrofit,* noe vi tidligere har beskrevet hvordan tas i bruk i rapporten. Under filen *MapFragment.kt,* hvor vi finner *onCreateView()*-metoden finner vi et *Retrofit*-kall. Her hentes data om de diverse stasjonene Meteorologisk Institutt har tilgang til i Norge. Disse lagres deretter i en liste-variabel vi kalte *temporarylist,* dette er fordi vi sender dem videre inn i en global-liste som står utenfor *Retrofit*-kallet. Herfra hentet vi fremvisningen av Mapbox ved å kalle på *getMapAsync* på et view av typen Mapbox. Inne i den åpnes muligheten for tilpasning av det visuelle ved kartet. Her ble blant annet kartet satt til å følge en av Mapbox sine visuelle utseender ved å kalle på funksjonen *mapbox.setStyle().* Mapbox har seks standard/innebygde stiler til kartene sine, og vi kan enkelt endre imellom disse ved å sette stilens URL som parameter i funksjonen nevnt ovenfor. Vårt valg falt på *MAPBOX_STREETS,* som er en allsidig stil som legger vekt på lesbar utforming av vei- og transittnett.

Videre lager vi en klasse av *MapAdapter*. Her har vi, som parametere, listen av stasjoner fra *Retrofit*-kallet sammen med en instans av Mapbox-kartet og *context*fra selve map-fragmentet. I *MapAdapter*-klassen gjøres enda et *Retrofit*-kall. Men til forskjell fra det første kallet som henter informasjon om stasjoner, gjør vi her i stedet et kall med data fra stasjonene og henter videre luftkvalitetsdata. Hovedproblemet her har vært rundt nestede *Retrofit*-kall og måten de kjøres på. Slik vi har valgt å vise frem data på er gjennom å fremstille de forskjellige stasjonene til Meteorologisk Institutt. Disse stasjonene hentes for seg selv i et eget *Retrofit*-kall hvor kallene i hovedsak kun returnerer stasjonsnavn med medfølgende koordinater. Som man enkelt skjønner vil ikke dette i seg selv være nok til å fremstille luftkvaliteten. Derfor måtte vi med disse resultatene hente tilhørende luftkvalitetsinformasjon gjennom et nytt *Retrofit*-kall. Vi begynte første med å hente stasjoner og deretter brukt dataene derfra til å kalle på "/", som er hoved forespørselen for luftkvalitetsdata. Denne forespørselen har, slik

det står beskrevet i meteorologisk institutts nettside, seks forskjellige parametere. Blant disse er koordinatene, og derfor krevde det at vi første gjorde kall på stasjoner.

En ting ved *Retrofit* var at man kunne kjøre kall både asynkront og synkront. I vårt tilfelle ville synkrone kall føre til at hovedkallet på data ville gi nullverdier. Dette var fordi det første kallet ikke hadde rukket å kjøre igjennom før det neste kallet skulle bli kjørt. Løsningen på dette var å gjøre koden i to deler, ved *MapFragment* og *MapAdapter*. *MapFragment*-klassen fungerer som hovedklassen i kart-funksjonen hvor kartet blir initialisert og hvor første *Retrofit*-kallet gjøres. Mens *MapAdapter*i stedet fungerer som en hjelpe-klasse. Fra hovedklassen sender vi stasjons-koordinatene inn i *MapAdapter*-klassen, og videre gjør vi nye kall og henter luftkvalitetsdata. For å hente dataen som er mest oppdatert og nærmeste det gjeldende tidspunkt bruker vi en *getCurrentTime*-metode som henter tidspunktet fra mobilen. Med hjelp fra denne kunne vi derfor sette riktig *AQI*-nivå til hver av stasjonene. Videre tok vi i bruk enkle if-else-spørringer til å ta i bruk riktige ikoner i forhold til *AQI*-nivå. Disse ikonene har vi tatt utgangspunkt fra Androids egne vektorikoner, men har valgt å modifisere de ved å legge til sorte outlininger samt sette de i de fire gjennomgående fargekodene vi bruker for å representere nivå. Deretter kunne vi enkelt fremstille hvert av stasjonene, med tilhørende *AQI*-nivå, med passende ikoner gjennom funksjonen *addMarker().* Denne brukes enkelt ved å legge til markørens tittel, beskrivelse, posisjon som parametere.

4.4 Liste over kommuner

For å opprette listen blir det brukt 3 *Kotlin* filer: *CountyFragment.kt*, *CountyLocationActivity.kt* og *ListAdapter.kt*. Det blir også tatt i bruk av et Interface *GetCountyService* som kobler seg til *API*-et for alle kommuner, samt klassene *CountyDTO*, som tar vare på informasjon fra hver kommune hvor de viktigste er navn og koordinater. Vi har også klassen *MyLocationDataDTO* som tar vare på data fra hver lokasjon basert på koordinatene vi har fra *CountyDTO* for bruk senere.

Listen over kommuner blir bygget opp av to filer, hvor den ene heter *activity_countylist.xml*. Denne filen inneholder et relativeLayout[22] som har iden *body*, og er sentrert med høyde og

[22] RelativeLayout = en viewgruppe som viser viewets «barn» i relative posisjoner

bredde satt til *wrap_content* [23]og *match_parent*[24]. Inne i *body* er det et *RecyclerView*[25] som er implementert i gradel fra biblioteket *"com.android.support:recyclerview-v7:28.0.0"*. Den andre filen har navne *layout_listitem.xml* og denne filen består av designet til hvert enkelt element som listen består av. Filen inneholder en relativeLayout med id *parent_layout* som har bakgrunn *textviewshadow* fra drawable mappen. Denne bakgrunnen gir ytterkantene en liten gråtone, slik at det blir lett å se hvor elementene er, og hvor man skal trykke for å treffe riktig element. Under *parent_layoyt* består filen av to *textView*. Det ene heter *txName* og skal inneholde navnet til kommunen. Fargen er satt til svart, teksten er sentrert og det er passet på at *textViewet* er satt helt til venstre i *parent_layout*. Det andre *textViewet* inneholder *AQI* nivået kommunen har på dette tidspunktet og har derfor fått iden *tvAQILevel*. Her er bakgrunnen satt til *textViewCercel* fra drawable mappen. Dette er en hvit avrundet bakgrunn, som skal bli til den fargen som representerer det nåværende *AQI* nivået til kommunen. Tekstfargen er hvit så man kan lett se hva som står der når det kommer en bakgrunnsfarge, typen er fet skrift og det er passet på at textViewet er plassert helt til høyre i *parent_layout.*

I *KommunerFragment.kt* filen blir det først opprettet to arrayer, *counties_string* er en liste med strenger som inneholder navnene på de kommunene som skal vises i listen. Det blir også opprettet en liste, *counties_choosen*, som er en liste bestående av elementer *CountyDTO*. Deretter blir *onCreateView()* funksjonen overskrevet og *counties_string* blir opprettet ved en funksjon *getCounties()*. Denne funksjonen sjekker om det finnes en fil som heter CountyList.txt. Hvis denne filen ikke finnes opprettes den og legger inn et utvalg av norske kommuner til filen. Hvis filen finnes leser den ut et og ett element og legger det til i listen.

Deretter settes en verdi for viewet til filen activity_kommunelist.xml fra layout. Denne filen inneholder et *recyclerView* som fylles inn litt senere. Etter opprettes et kall på *API*-et ved *Retrofit* på samme måte som beskrevet tidligere. Vi finner deretter det rette *recyclerViewet* fra viewet og setter layoutManageren til linear. Så utføres kallet på *API*-et som henter ut alle kommuner og for hver kommune fylles ut informasjonen i *CountyDTO* og settes inn i en ny liste som inneholder alle kommuner satt inn i hvert sitt *CountyDTO*. Når alle kommunene er opprettet i listen vil vi gå igjennom den nye listen for å sjekke om navnet til hver av kommunene finnes i listen med strengene som forteller hvilke kommuner som skal vises. Hvis

[23] Wrap_content = gjør at viewet holder seg til ytterkantene av innholdet
[24] Match_parent = setter viewet til samme størrelse som «foreldre» viewet
[25] RecyclerView = et view som er laget for å genere en liste med flere elementer, basert på layout fra en xml fil

kommunen har et navn som er i streng listen vil denne kommunen bli lagt til i listen *counties_choosen* med alle kommunene som skal vises. Når dette er utført opprettes det en *adapter*[26] som skal fylle ut informasjon og sette designet til hvert element som puttes inn i *recyclerViewet*. Her sender vi med *counties_choosen* listen før vi til slutt returnerer viewet.

ListAdapter.kt filen er til for å generere hvert enkelt element i *recyclerViewet*, og vi tar inn listen med valgte kommuner som argument, ettersom det er disse kommunene som skal bli vist til brukeren i systemet. Ettersom *ListAdapter* implementeres som en *ReyclerView* adapter kreves det at de tre funksjoner må implementeres og overskrives:

- *onCreateViewHolder*, oppretter en verdi for view
- *onBindViewHolder*, setter informasjon i viewet for hvert element
- *getItemCount*, returnerer antall elementer

Det første som skjer er at vi implementerer *onCreateViewHolder()* som kun setter en verdi for view. Denne verdien er den xml filen som inneholder designet til hvert enkelt element i listen og kommer fra filen layout_listitem.xml. Deretter tar vi for oss *onBindViewHolder()*. Her kaller vi på *Retrofit* for hvert element av *CountyDTO* som skal genereres, ved elementets lengdegrad og breddegrad, og vi vil dermed finne all informasjon for luftkvaliteten på dette stedet og legger denne informasjonen inn i *MyLocationDataDTO* for kommunen. Vi setter så navnet til kommunen på rett plass ved hjelp av en holder klasse vi kommer tilbake til litt senere. Så må vi sette riktig *AQI* nivå til den rette tiden, altså hva nivået er på akkurat nå. Dette gjør vi ved å finne ut hva tiden er med *getCurrentTime()* metoden som er beskrevet tidligere. Vi finner så den riktige tiden i listen med tider fra *MyLocationDataDTO* og setter teksten i systemet til å vise en verdi avrundet til to desimaler og setter bakgrunnen til teksten til den rette fargen i forhold til *AQI* nivået.

Den siste funksjonen som må implementeres er *getItemCount()*. Denne skal kun returnere størrelsen på listen *ListAdapter* tok inn, altså antall elementer som skal bli generert. Det blir også opprettet en klasse som er en sub klasse av *recyclerView* viewholder. Denne klassen lager en *onClickListener* for hvert element i listen og deretter oppretter en intent med lengdegrad, breddegrad og navn på kommunen slik at vi senere kan aksessere disse dataene til

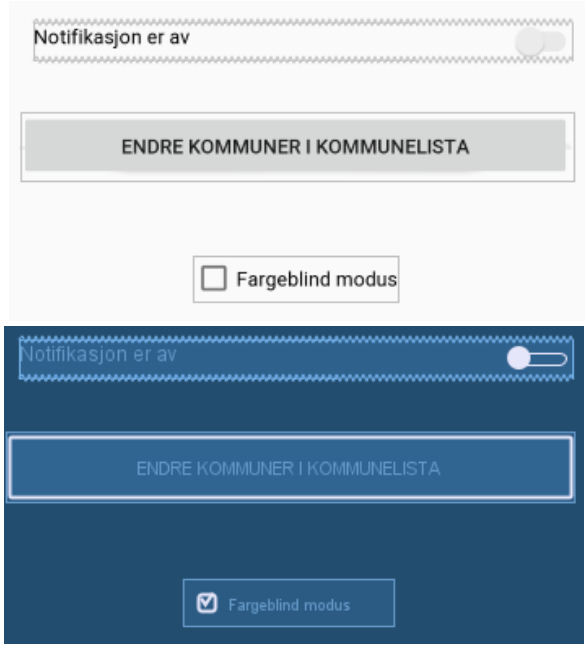
[26] Adapter = generer data for hvert element i et recyclerView

den kommunen brukeren trykker på, og dermed kunne kalle på riktig data fra *API*-et. ViewHolder definerer også *textViewet* for navn og *AQI*-nivå, som blir satt for hvert element.

Filen *CountyLocationActivity.kt* består av kode som er veldig lik *LocationFragment.kt*, med kun noen små endringer for å optimalisere brukeropplevelsen og håndtere hvilken lokasjon det skal kalles på. Det blir her aksessert data fra den intenten som ble trykket på av brukeren, og man setter da lengde og breddegrad for denne kommunen, for å så kalle på data fra denne lokasjonen. I tillegg er designet i *activity_countylocation.xml* litt annerledes for å gi brukeren følelsen av at han/hun ikke er på denne lokasjonen.

4.5 Innstillinger

Innstillinger-funksjonaliteten har et enkelt design bestående av en bryter for å skru av eller på notifikasjoner. Bryteren vil vise en dialog som ber brukeren skrive inn informasjon om når han/hun ønsker en notifikasjon, og en knapp for å bekrefte. Det vil deretter være gjenkjennelig hvilke betingelser man har valgt for varslingen på siden. Det vil stå «Notifikasjon ved …», og bryteren vil ha en grønn farge. For å skru av notifikasjoner, trenger man bare å trykke på bryteren en gang til, og godkjenne at man faktisk ønsker å skru av varsling. Bryteren vil ved oppstart være avslått, og ha teksten «Notifikasjon er av.» Når betingelser er møtt og notifikasjon dukker opp, vil vi se at temafargen bestemmes av de generelle fargekodene for *AQI* i systemet. Hvis *AQI* for eksempel er under 2, vil teksten være grønn.



16 Innstillinger sett fra design perspektiv

I tillegg til bryteren for varsling finnes en knapp som fører brukeren til en helt ny side. Denne siden gir muligheten til å endre elementene man finner i listefunksjonen. Dette har vi fått til ved hjelp av dialoger som forklarer og viser brukeren hva han skal skrive og velge, samt knapper som beskriver hva som skjer dersom man trykker på dem. Ettersom «?» og «+»

henholdsvis er universelle tegn for «hjelp» og «legg til», vil disse knappene også være lett forståelige og gjenkjennelige. Vi har på denne måten klart å gjøre dette så minimalistisk og brukervennlig som mulig, slik at brukere enkelt har mulighet til å orientere seg i innstillingene. Det blir derfor også en lav læringskurve for brukere på alle nivåer av erfaring. Den siste funksjonaliteten er en enkel checkbox, som skrur av eller på fargeblind-modus for alle funksjonalitetene i systemet. Den viser en «check» dersom modusen er på, og er tom ellers.

Innstillingene benytter seg av fire Kotlinfiler, som alle er samlet i en mappe kalt «Settings». Den har også tilleggs funksjonaliteter i *CountyFragment.kt* og *MainActivity* som blir forklart mer i detalj senere. Notifikasjonen opprettes i *SettingsActivity.kt* og *NotificationService.kt*. Dersom en bruker ønsker å opprette notifikasjon, vil han få valget å skrive inn en kommune i et felt. På forhånd gjøres et kall til *API*-et som fyller en adapterliste med alle kommunenavnene det er mulig å velge fra. Denne listen består av name-strengene fra *CountyDTO* -lista som returneres i kallet. I det en bruker skriver inn bokstaver i feltet, vil den da vise forslag på kommunenavn som begynner på bokstavene skrevet inn. Dette gjøres ved en *CompleteTextView*[27] som har en adapter med en liste med alle kommunestrengene. Hittil er det forventet at brukere er noenlunde kjent med hvordan *AQI*-nivåer opptrer, så bruker vil derfor få en liten liste med forhåndsbestemte *AQI*-verdier å velge i. Disse vises i en «Spinner» av «Double»-verdier. Dersom et kommunenavn er skrevet feil, eller rett og slett ikke eksisterer i *API*-et, vil en *Toast* si ifra om at noe galt skjedde ved opprettelse av notifikasjon. Når riktige verdier er skrevet inn i begge feltene og «Opprett plan for varsling»-knappen er trykket, vil to viktige endringer skje i systemet. Det er nemlig viktig at endringene som har skjedd i innstillingene forblir endret selv når systemet avlsuttes. Dette problemet løses ved å opprette en lokal *PreferenceManager*[28], som kan lagre verdier internt på enheten, og hente det ut igjen med enkle metoder. Verdiene lagres deretter i *SharedPreferences*[29]. For at bryterens tekst og tilstand skal forbli lagret, må vi derfor gjennom en *PreferenceManager* lagre *Boolean*-verdien til tilstanden og *String*-verdien til beskrivelsen. Disse hentes også frem igjen når man åpner innstillinger på nytt ved et senere tidspunkt. Deretter opprettes en plan for varsling.

[27] CompleteTextView = TextView som viser forslag til hva bruker kan skrive inn
[28] PreferenceManager = Interface for lagring og uthenting av verdier lokalt på enheten
[29] SharedPreferences = Der alle verdier som lagres internt på enheten lagres

For at varslingen skal fungere i bakgrunnen av appen, ble vi nødt til å utnytte en *jobScheduler*[30] i metoden *scheduleJob*. Denne planlegger kodesnutter som skal kjøre i et tidsintervall gitt at enheten har tilgang til internett. Som forklart tidligere, valgte vi at den skulle kjøre kodesnutten hver 6. time. Det blir deretter opprettet en instans av klassen *NotificationService*, som finnes i *NotificationService.kt*. Den importerer også grensesnittet *JobService*[31], som består av tre metoder; *onStartJob*, *doBackgroundWork* og *onStopJob*. *onStartJob* tar inn parameterne for varsling via en *PersistableBundle*, ettersom metoden ikke tar inn parametere på vanlig måte. Disse parameterverdiene lagres i klassen. Deretter kjøres metoden *doBackgroundWork*. Metoden utfører først et kall til *API*-et ved metoden *callAPI*, og henter ut- og setter verdi for den momentane *AQI*-verdien i den valgte kommunen. *doBackgroundWork* sjekker deretter om det målte *AQI*-nivået er høyere enn valgt *AQI*-nivå. Hvis det skulle stemme, vil programmet via en *NotificationManager*[32] opprette en notifikasjon, som beskriver hva *AQI*-nivået målt er for en gitt kommune. Hvis *AQI*-nivået er lavere enn satt verdi, vil ingenting skje. Hvis enheten mangler internett-tilgang, vil den heller ikke kunne få tilgang til *API*-et. Metoden *onStopJob* vil da kjøre og sette arbeidet på pause inntil enheten igjen får tilkobling til internett. På grunn av limitasjoner på eldre programvare, var vi nødt til å begrense hvilke *API*-nivåer systmet kan kjøre på. *NotificationManager* har to versjoner som enten bare kan kjøre på *API* høyere enn 26, eller lavere enn 26. Vi prioriterte derfor nyere programvare. Den største forskjellen er at varsling i *API* høyere enn 26 krever en kanal å kjøre på. Denne kanalen har id *CHANNEL_1_ID*, og opprettes faktisk i det systemet åpnes for første gang i *MainActivity.kt*. Metoden *createNotificationChannel* oppretter denne kanalen, slik at notifikasjoner faktisk skal virke på enhetene. Dersom alt har gått riktig for seg, vil notifikasjoner dukke opp på enheten som planlagt.

CountyListActivity.kt er aktiviteten som kjøres dersom man trykker på «endre kommuner i kommunelista»-knappen. Aktiviteten er veldig lik den i listefunksjonen i det at den består av et *RecyclerView* som inneholder navnene til kommunene man har i lista. Klassen er nødt til å gjøre et kall til *API*-et for å fylle en liste med alle fylkene i Norge. Dette kallet er helt likt kallet som gjøres når man skal skrive inn kommunenavn ved opprettelse av varsling. Den

[30] JobScheduler = Interface som planlegger jobb som skal utføres i bakgrunnen
[31] JobService = Interface for å gjøre klar en jobb som skal utføres i bakgrunnen
[32] NotificationManager = Interface for å opprette en notifikasjon som skal vises på enheten

brukes også på samme måte ved at den viser forslag på kommunenavn i det en bruker begynner å skrive. For redigering av lista benyttet vi en *DragManageAdapter*[33], som via metodene *onMove* og *onSwiped* henholdsvis flytter på- og sletter et element fra lista. Lista har en *CountyListAdapter* i *Kotlin*filen med samme navn, som har metoder som *removeItem, addItem, swAPItems* og *setCounties*. Klassen tar inn kontekst og elementlisten som parametere, slik at den kan behandle listen. *removeItem* og *swAPItems* tar inn posisjonen elementet er (og evt. flyttes til) og fjerner det, eller flytter det i listen. *addItem* tar inn kommunenavnet og legger til elementet på slutten av lista. Ved enhver endring på lista oppdateres viewet med de nye endringene, slik at bruker enkelt kan se resultatet. *setCounties* er en metode som hjelper til med lagring av verdiene i lista. Ettersom vi bruker en *ArrayList*, er det ikke mulig å bruke en *PreferenceManager* for lagring av lista, slik vi gjorde tidligere. Vi var derfor nødt til å skrive til en intern fil på enheten via en *BufferedWriter* bestående av en *FileWriter*[34]. Slik var det mulig å lagre hele lista med alle elementene gjennom en *for*-løkke. Denne filen, kalt *CountyList.txt* vil lagres selv når systemet lukkes. I Listeaktiviteten beskrevet tidligere vil metoden *getCounties* hente ut verdiene fra filen. Dersom filen ikke er opprettet fra før, for eksempel ved første gang man åpner systemet, vil den selv opprette filen og fylle listen med et utvalg av de største kommunene i Norge. Hvis filen derimot allerede finnes, vil den benytte seg av de kommunene som finnes i filen, og skrive dem ut på en lett og forståelig måte.

Fargeblind-modusen er gjennomgående i alle funksjoner i systemet. Det finnes nemlig alternative fargeblind-vennlige filer for flere av XML-filene systemet utnytter. Det eneste sjekkboksen i settings-aktiviteten gjør er å sette en *boolean*-verdi i en *SharedPreference*, som de andre aktivitetene lett kan hente frem igjen. Ved enkle if-spørringer i alle aktivitetene kan systemet skille mellom hvilke farger den skal vise når aktivitetene kjøres.

4.6 Navigasjons meny

Funksjonaliteten for navigasjon har et ganske enkelt design, og består av 4 meny-elementer som kan trykkes på. Menyelementene vil vise ulike fragments ut ifra hvem som blir trykket på, men fortsatt beholde hele navigasjons meny nederst i systemet. For å kunne gjøre dette brukes *BottomNavigationView* og *FrameLayout* for å vise de ulike fragmentene. Filene som

[33] DragManageAdapter = interface for adapter som tar for seg hva som skjer dersom man drar listeelementer i ulike retninger
[34] FileWriter = Klasse som skriver til fil

blir brukt heter derfor «...Fragment», og ikke Activity. Fragment er en slags sub-activity. Filene arver derfor Fragment og ikke *AppCompatActivity*. For å kunne vise de ulike fragmentene brukes det *FrameLayout* i *activity_main.xml* denne har fått id *rootLayout.* Denne består av resten av *viewet* i *activity_main.xml*. Det er her hvert fragment blir vist når de blir lastet opp av et element i menyen.

I Android's design support bibliotek versjon 25 er *BottomNavigationView* en del av dette biblioteket. For å hente *BottomNavigationView* må vi inkludere design support biblioteket i systemets *build.gradel* fil ved å legge til følgende: *implementation 'com.android.support:design:25.0.0'. BottomNavigationView* er blitt implementert inn i vår *activity_main.xml*. Vi har gitt denne viewId *navigation.* Så har vi definert noen meny elementer som skal settes inn i vår *navigasjon*. Dette gjøres i *res->menu->navigation.xml*. Meny elementene vi har er 'Lokasjon', 'Kart', 'Liste' og 'Innstillinger'. Alle elementene har hver sin drawble icon som ligger i *res->drawable.* Hver meny element henter frem og starter et nytt fragment.

For at menyelementene skal funke og vise oss de ulike fragmentene må vi deklarerer en listener som kan håndterer dette. Variabelen blir kalt *OnNavigationItemSelectedListener.* For å lage denne listener bruker vi *BottomNavigationView.OnNavigationItemSelectedListener*. Inni denne listeneren lager vi først en variabel vi kaller *fragment* av typen *Fragment,* og setter den lik null foreløpig. Så brukes det *when* utrykket til å sjekke hvilken meny element som er blitt valgt. For hver meny element som blir valgt «starter den» riktig fragment ved hjelp av en statisk metode i hver fragment klasse *newInstance().* Dette blir gjort i *Kotlin* ved hjelp av companion objects som lar deg deklarere variabler og funksjoner knyttet til klassen. Slik som statisk metode i Python. Menyelementet deklarer så den til variablen *fragment* som ble opprettet i starten av listeneren. Etter det blir gjort avslutter *when* og en rekke kommandoer blir kjørt for å kunne erstatte eller sette inn fragmenten. Først henter vi frem *FragmentManager* [35]ved *supportFragmentManager.* For at det skal kunne bli gjort noen endringer trenger vi så å hente *FragmentTransaction* [36]slik: *supportFragmentManager.beginTransaction().*

[35] FragmentManager = Interface for samhandling med Fragmentobjekter inne i en Activity
[36] FragmentTransaction = API for å utføre et sett med Fragment operasjoner

I *FragmentTransaction* er det en rekke metoder, men vi bruker *replace(int containerViewId, Fragment fragment)*. I denne sender vi med viewId til der hvor fragmenten skal settes. Vi vårt tilfelle er dette FrameLayout som vi ga id *rootLayout*. Så må vi også sende med fragmenten som ble laget/startet av meny elementet, altså *fragment*. Tilslutt vil vi få en linje som ser slik ut: *supportFragmentManager.beginTransaction().replace(R.id.rootLayout, fragment!!, fragment.javaClass.simpleName).* Denne kan vi nå kjøre *commit()* på så vil viewet i FrameLayout endres til det brukeren trykket på av meny.

5 Testdokumentasjon

Måten vi som gruppe valgte å utføre testingen av systemet var å teste implementasjoner og funksjonaliteter regelmessig under utviklingsprosessen. Dette er fordi vi ønsket å jobbe i mindre partier og samtidig få en mer fleksibel måte å utvikle på fremfor en mer plandrevet utviklingsstil. Vi hadde to ukentlige møter hvor vi i det første møtet, i starten av uken, delte ut oppgaver til hver enkelt av oss og i det andre møtet, på slutten av uken, viste frem det vi hadde kommet frem til eller eventuelt opplyste hverandre om problemene som hadde oppstått. Som følger av denne måten å dele inn oppgaver til hver enkelt, resulterte det i at vi raskere fikk se resultater. Og i tillegg kunne vi kjappere oppdage diverse problemstillinger og error i koden, som gjorde slik at vi på nytt kunne ta nye vurderinger til funksjonaliteten som var under utvikling. Et eksempel på dette var innad i Mapbox hvor vi valgte å la være å implementere *SymbolLayer*. Vi vurderte det slik at å ta i bruk simplere markører var bra nok i forhold til hvor mye ressurser som krevdes for å ta i bruk *SymbolLayer*-klassen i vårt system.

Med testingen hadde vi i tankene om å se om de funksjonelle kravene vi på forhånd hadde satt faktisk ble implementert og fungerte. Om dette ikke var tilfellet gikk vi frem med debugging. For å få til enhetstesting, som betyr å teste individuelle biter av koden, brukte vi i hovedsak Android Studios innebygde debug modus. Debugging kan man si er prosessen i å finne og løse feil eller uforventet oppførsel i koden. Det kan være alt i fra at for eksempel lokasjonssiden henviser til feil lokasjon eller til at systemet krasjer. Debuggeren fungerte på den måten at man velger å plassere breakpoints, som er et punkt i koden der kjøringen av systemet pauses ved det gitte punktet eller linje i koden. Derfra fikk man mulighet til å undersøke og evaluere variabelendringer og oppførsel i koden, som man kunne gå videre fra linje for linje. Et annet hjelpemiddel som vi opplevde som svært viktig for enhetstesting var å skrive forskjellige *Logcat*-meldinger i koden. Med hjelp av dette kunne vi se om kjøringen for

eksempel nådde frem eller ga riktig verdi, siden disse meldingene enten ville komme direkte opp i terminalen eller ikke ettersom om koden ble eksekvert.

5.1 Sikkerhet

Under utviklingen av et system spiller sikkerheten nesten alltid en viktig rolle. I vårt system lagrer vi ikke noe av brukerens sensitive informasjon. En eventuell angriper har derfor ikke så mye å hente på den fronten. Vi prøvde å lage ulike angriperhistorier, og trussel scenarioer. Vi kom frem tilslutt at den største og eneste trusselen var angrep mot API-et igjennom vårt system. En eventuell hacker kan endre tidspunktet på enheten og lokasjon flere ganger i sekundet, som ville ha sendt flere kall på API-et og potensielt fungere som en DDoS. For å kontre dette har API-et vi bruker egne rettingslinjer for dette, og vil nemlig sperre systemet som sender for mange kall på kort tid. For å likevel hindre at et slikt angrep er mulig gjennom systemet vårt kan man sende med noen verdier når vi kaller på metoden *requestLocationUpdates()* som igjen gjør et kall på AP-Iet. Det er *minTime* og *minDistance*. Verdiene, er nevnt i produktdokumentasjonen, som vi sender med gjør det nesten umulig for en hacker å bruke vårt system for å DDoS-angripe API-et.

5.2 Universell Utforming

Under utvikling av systemet ønsket vi at så mange som mulig skulle ha mulighet til å bruke systemet vårt. Det var derfor viktig at vi fokuserte på Universell Utforming[37] gjennom hele utviklingsprosessen Det første vi fokuserte på i designet var å ha tykk skriftstørrelse som stod i kontrast til bakgrunnen. Flere steder har vi derfor benyttet svart tekst på hvit bakgrunn, eller hvit tekst på farget bakgrunn. På denne måten skaper vi få problemer med svaksynte brukere av systemet. Vi utnyttet også klare og forklarende farger slik at de fleste brukere skal ha et forståelig møte med systemet. Fargeblindhet blir derfor et problem for en potensiell bruker av systemet vårt. For eksempel vil det for dem være vanskelig å skille mellom rød og grønn farge. Derfor tok vi tiden til å lage en fargeblind-modus. Denne modusen endrer alle de forklarende fargene til forskjellige grå- og blåtoner. På denne måten har vi gjort systemet brukbart selv for fargeblinde brukere.

[37] Universell Utforming = Praksisen å gjøre et system tilgjengelig for alle, på tross av potensielle funksjonshemninger

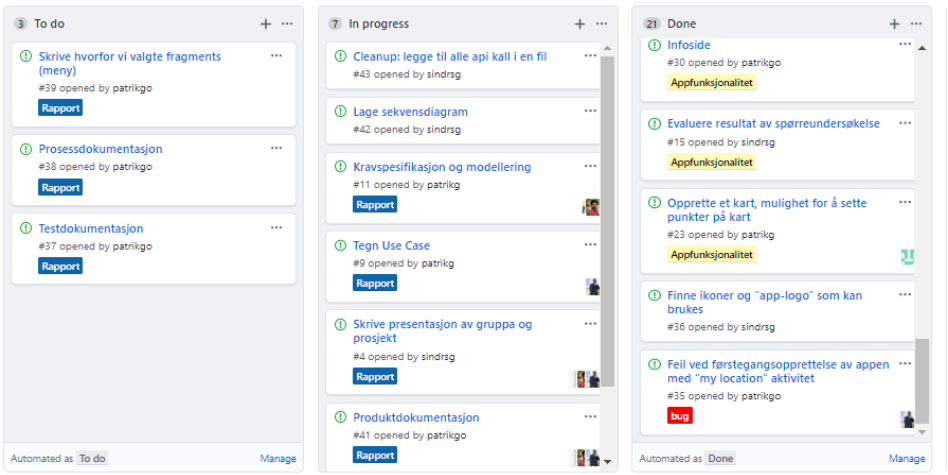
6 Prosessdokumentasjon

Prosessen som har ledet frem til det endelige produktet har gått bedre enn hva vi alle hadde forventet. En fordel vi hadde som gruppe var at vi alle kjente hverandre fra før av, som gjorde samarbeidet og kommuniseringen mellom oss mye lettere og bedre. Men ved at vi kjente hverandre fra før kan også ha brakt negative sider, blant annet at vi kanskje har gitt hverandre litt for frie tøyler til tider. Når det kom til krav av oppmøte var vi ikke så kritiske, som vi kanskje burde vært, mot hverandre. Siden vi ikke hadde noe tidligere erfaring med prosjektarbeid og utvikling av et system i denne størrelsen, var vi litt usikre på hvor vi skulle begynne. Vi visste at vi ville ta i bruk scrum som utviklingsmetodikk, derfor valgte vi i likhet med metodikken scrum å starte med planleggingsfasen.

Vi visste at vi måtte bruke Android Studios med *Kotlin* og github, dermed begynte vi å se og lære oss hvordan dette fungerte. Vi ønsket at alle hadde god kjennskap til github, derfor brukte vi en del tid på å passe på at alle var på samme bølgelengde når det kom til dette. Mens vi holdt på med dette lagde vi brukerhistorier med funksjoner og ideer vi ønsket å ha med i systemet eller som passet til caset vårt. Funksjonalitetene vi endte opp med var basert på tipsene vi fikk i casebeskrivelse og egne ideer. Da vi ble ferdig så vi tilslutt at vi satt igjen med alt for mange funksjonaliteter i forhold til tiden vi hadde å bruke.

Ettersom vi allerede hadde planlagt å lage en spørreundersøkelse, valgte vi også å spørre potensielle brukere om viktigheten ved alle funksjonene, og ut fra dette fikk vi en pekepinn på hva som til slutt kanskje ville bli de mest essensielle funksjonalitetene i systemet. I spørreundersøkelsen var vi samtidig interesserte i å vite hvor attraktivt et slik systemet er blant folk generelt. Spørreundersøkelse hjalp oss til å finne en mer konkret målgruppe vi skulle rette systemet mot. Men ikke minst var det interessant å vite om noen brukte andre systemer for å se luftkvaliteten, og i så fall hvilke de brukte.

Da vi var vi ferdige med planleggingsfasen satt vi igjen med en scrum-backlog. Ved start av gjennomføringsfasen fant vi ut at sprint og sprint-backlog ikke var helt egnet til slik vi ønsket å arbeide, ettersom vi alle hadde andre prioriteringer i tillegg til dette prosjektet. Vi fant derfor ut at vi skulle ta i bruk en hybrid versjon med scrum og kanban, scrumban. Derfor endte vi opp med å lage et kanban-board i git repositeriet vårt under prosjekt.



17 Kanban board i git repo

Under gjennomføringsfasen er det spesielt viktig å ha ulike møter underveis. Siden vi i utgangspunktet ønsket å følge scrum metodikken. Vi planla å implementere smidige praktiserer i møtene våre, men fant fort ut at de klassiske scrum sprintene ikke virket for oss. Likevel hadde vi standup-møter som varte i ca. 15 min hver mandag, men holdt kommunikasjonen gående gjennom hele prosjektet. Resten av mandagsmøtet brukte vi til å presentere nye ideer og videreutvikle systemet. Frister for når arbeidet skulle vært ferdig satt vi mot slutten av møte. Mandagsmøtene varte som regel fra 10:00 til 14:00 hver uke.

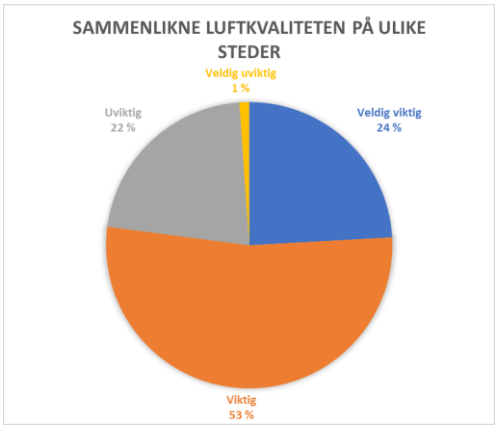
I tillegg til møte på mandag hadde vi også et møte hver fredag. I starten av møte hadde vi en gjennomgang på hva hver enkelt hadde gjort siden sist. Dette funket som en sprint gjennomgang. På denne måten klarte alle å holde følge med utviklingen og fristene vi hadde satt. Vi oppdaget derfor også tidlig om noen satt fast eller trengte hjelp. Resten av tiden på fredagen gikk til å ha et retrospektivt møte. Da hadde vi en gjennomgang hvordan hele uken gikk, hva som gikk bra, hva som gikk dårlig og hva som burde endres på. Vi prøvde hele tiden å finne forbedringspunkter med utgangspunkt i hva vi har arbeidet med den siste uken. I likhet med mandagsmøtene hadde vi satt av fra 10:00 til 14:00.

Gjennom utviklingen hadde vi fire hoved-funksjonaliteter som vi ønsket systemet skulle ha, men igjen var det flere små funksjonaliteter under disse. Hvert medlem i gruppen fikk dermed ansvaret for hver sin funksjonalitet. Gjennom møtene skulle hvert medlem oppdatere resten av gruppen på hva og hvordan dette hadde blitt gjort. Ettersom at vi aldri har gjort ting før, valgte vi å gjøre det slik for at alle skulle kunne lære mest mulig, men også fordi vi fant det mest effektivt. Noen ganger var det også behov for å bruke hverandres løsninger og igjen ble

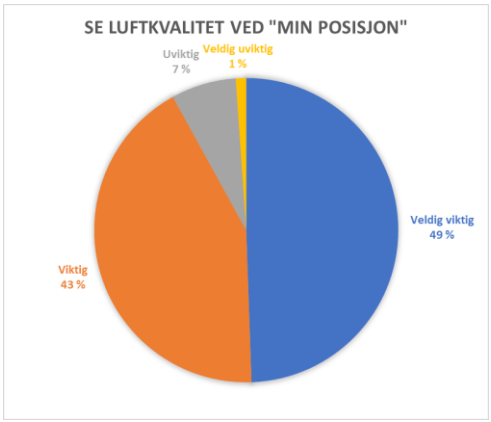
det viktig at alle hadde god forståelse for hverandres kode. Dette hjalp senere i utviklingen da det skulles skrive en rapport på systemet. Om det skulle oppstå problemer underveis eller at en av oss satt fast kunne vi enkelt hjelpe hverandre ettersom alle visste hvordan de diverse funksjonalitetene funket.

Mot slutten av prosessen, altså avslutningsfasen, gikk vi igjennom hver enkelt funksjon til systemet og diskuterte mulige forbedringer. Det var også stort behov for opprydning og optimalisering av kodene. Ettersom det var første gang vi programmerte i *kotlin* var ikke koden helt optimal i starten, men utover i læringsprosessen lærte vi nye og bedre løsninger på kode, som gjorde at det ble litt forskjeller i kodestil gjennom prosjektet.

I avslutningsfasen av prosjektarbeidet kom vi frem til at vi måtte nedprioritere og velge bort enkelte av de funksjonelle kravene vi hadde satt. Avgjørelsen på hva vi skulle prioritere bort var hovedsakelig basert på spørreundersøkelsen, men også basert på tidsforbruk. Kravspesifikasjonene ved figur 18 nedenfor var et eksempel på en funksjon som endte opp med å bli bortprioritert på bakgrunn av undersøkelsen, ettersom de fleste mente dette var uviktig, samt få som mente det var veldig viktig. Spørreundersøkelsen sa også at luftkvaliteten ved min posisjon var viktig, og derfor ble dette vår hovedprioritet som vi til slutt valgte å ha som forside. Se figur 19.



18 En funksjonalitet vi valgte bort, basert på spørreundersøkelsen



19 Den mest etterspurte funksjonaliteten, basert på spørreundersøkelsen

For å konkludere har prosessen vært både lærerik og utfordrende. Vi har måttet lære å jobbe i team på en helt annen måte enn tidligere, og vi har skaffet oss god erfaring når det kommer til arbeidslivet. Arbeidet har gitt oss et godt innblikk i forskjellige prosjektarbeidsmetoder og vi har fått prøvd ut noen av disse i praksis. Selv om vi er godt fornøyde med hvordan systemet

endte opp, føler vi også at vi ønsket flere funksjonaliteter og dette er noe vi eventuelt ville implementert ved senere eventuelle oppdateringer. Under gjennomførelsen har vi hele tiden prøvd å holde arbeidet profesjonelt og faglig, men med balansering av blant annet andre fag har vi til tider sklidd ut. Når vi ser tilbake på arbeidet, ser vi at det hadde vært lurt å gjøre enda flere undersøkelser ved oppstart, som intervjuer og undersøkelser. I tillegg ser vi at det ikke har blitt gjennomført testing av systemet med potensielle brukere, annet enn oss selv, noe som i utgangspunktet er svært viktig under utvikling av systemer. Hvis vi hadde inkludert dette hadde vi fått enda bedre tilbakemeldinger under arbeidsprosessen fra de som faktisk skulle bruke systemet. Til tross for dette har oppgaven vist stor nytteverdi for enhver i gruppen, og vi er både stolte og imponerte av egen innsats og resultatet.

7 Bibliografi

Rubin, K. S. (2013). Essential Scrum: A practical guide to the most popular agile process. Upper Saddle River, NJ: Addison-Wesley.

Folkehelseinstituttet. (2017). Luftforureining I Norge . Hentet fra https://www.fhi.no/nettpub/hin/miljo/luftforureining--i-noreg/

Jungleworks. (u.å.). 5 things to know before choosing digital maps *API*: Google Maps Vs Mapbox explained. Hentet 14. mars, fra https://jungleworks.com/google-vs-mapbox/

Arnesen, M. (22. Februar 2017). Kvinne hentet ned fra fjellhylle med helikopter. Hentet 30. mars, fra https://www.nrk.no/hordaland/kvinne-hentet-ned-fra-fjellhylle-med-helikopter-1.14497665

WMS – Web Map Service . (12. oktober 2018). *Store Norske Leksikon*. Hentet 10. mars 2019 fra https://snl.no/WMS_-_Web_Map_Service

Mapbox (u.å.). Maps *SDK* for Android. Hentet 14. mars 2019, fra https://docs.mapbox.com/android/maps/overview/

Objects and companion objects. (u.å.). Hentet fra https://kotlinlang.org/docs/tutorials/kotlin-for-py/objects-and-companion-objects.html

Making SharedPreferences Easy with Kotlin. (26. september 2016). Hentet fra https://blog.teamtreehouse.com/making-sharedpreferences-easy-with-kotlin

Square. (16 april 2019). Square/retrofit. Hentet fra https://github.com/square/retrofit

Baeldung. (04. november 2018). Writing to a File in *Kotlin*. Hentet fra https://www.baeldung.com/kotlin-write-file

Kmvignesh. (30. april 2018). Kmvignesh/LocationExample. Hentet fra https://github.com/kmvignesh/LocationExample

Indragni Soft Solutions. (10. februar 2018). How to Get Current Location Latitude and Longitude in Android Studio Example. Hentet fra https://www.youtube.com/watch?v=XQJiiuk8Feo&t=2s

Codeing in Flow. (24. februar 2018). BottomNavigationView with Fragments - Android Studio Tutorial. Hentet fra https://www.youtube.com/watch?v=tPV8xA7m-iw

Channel, E. (10. oktober 2018). Android *Kotlin* - Notification Example. Hentet fra http://www.youtube.com/watch?v=7NVpdzHVAQI&t=319s

CodeAndroid. (18. februar 2018). Android Tutorial (Kotlin) - 40 - Notification. Hentet fra https://www.youtube.com/watch?v=Fo7WksYMlCU&t=3s

Jones, B. (22. juli 2018). Use *Retrofit* to parse *JSON* in *Kotlin* Android app. Hentet fra https://www.youtube.com/watch?v=FW7sY7M_E8k

Diverse klasser vi har tatt i bruk

https://developer.android.com/reference/kotlin/android/location/LocationManager

https://developer.android.com/reference

https://developer.android.com/reference/android/support/v4/app/ActivityCompat.html#requestPermissions(android.app.Activity,%20java.lang.String[],%20int)

https://developer.android.com/reference/kotlin/android/location/LocationListener.html

https://developer.android.com/reference/java/util/Calendar

https://developer.android.com/reference/java/text/SimpleDateFormat

https://developer.android.com/guide/components/fragments

https://developer.android.com/guide/topics/ui/layout/recyclerview#structure

https://developer.android.com/reference/android/support/design/widget/BottomNavigationView

https://developer.android.com/reference/android/support/v4/app/FragmentManager.html

https://developer.android.com/reference/android/support/v4/app/FragmentTransaction.html

https://developer.android.com/reference/android/widget/FrameLayout

https://developer.android.com/reference/android/app/job/JobInfo

https://developer.android.com/reference/android/app/job/JobParameters

https://developer.android.com/reference/android/app/job/JobScheduler

https://developer.android.com/reference/android/app/PendingIntent

https://developer.android.com/reference/android/os/PersistableBundle

https://developer.android.com/reference/android/preference/PreferenceManager

https://developer.android.com/reference/java/io/File

https://developer.android.com/reference/kotlin/android/app/Notification.html

https://developer.android.com/reference/kotlin/android/app/NotificationManager