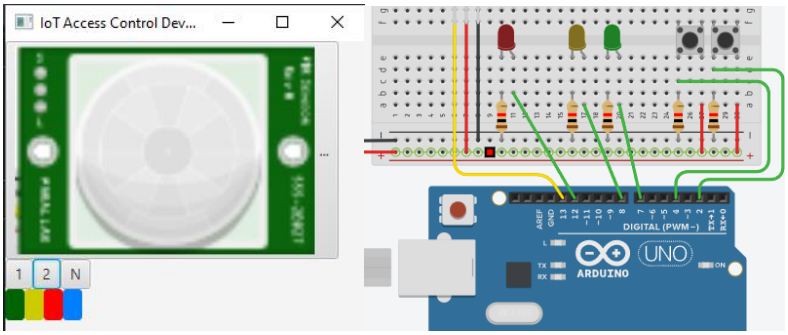
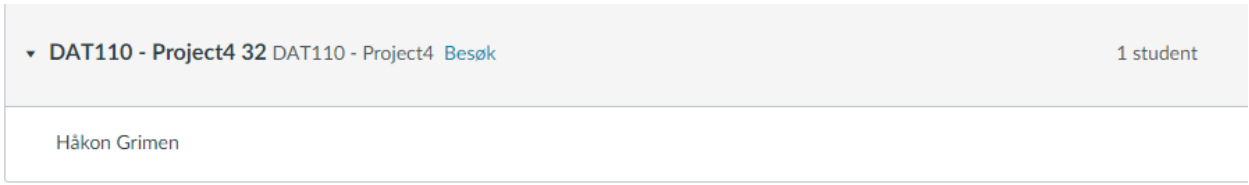
DAT110 – Prosjekt 4, våren 2022

An IoT-Cloud Access Control System

Håkon Grimen – studnr 131203

Canvas Gruppe: **DAT110 – Project4 32**





Introduksjon

TinkerCAD-designet finnes her:

https://www.tinkercad.com/things/7zEbwgT61Wg-magnificent-maimu-vihelmo/editel?sharecode=g3c5XYb0wKas5W_aHxGF04ZkwF7gc7ONZa-SV1e3Jb0

Det er også tilgjengelig som .brd fil på github, sammen med koden i en egen fil.

Koden til Part B finnes her:

https://github.com/h131203/dat110-project4-startcode-iotdevice
https://github.com/h131203/dat110-project4-startcode-cloudservice

Oppgaven her går ut på å designe et tilgangssystem. Systemet har en lås som åpnes og lukkes, avhengig av om du som bruker har tastet inn riktig kode eller ikke. I tillegg til det er det muligheter til å endre koden som kreves, automatisk loggføring av alle tilgangsforsøk (både godkjente og ikke-godkjente), samt søkemuligheter og opplisting av hele denne loggen.

Kode og logg er lagret i skyen, så systemet bygger på at IoT-enheten har kontakt med skyen. Dette gjøres via REST-APIet.

Access Control Design Model

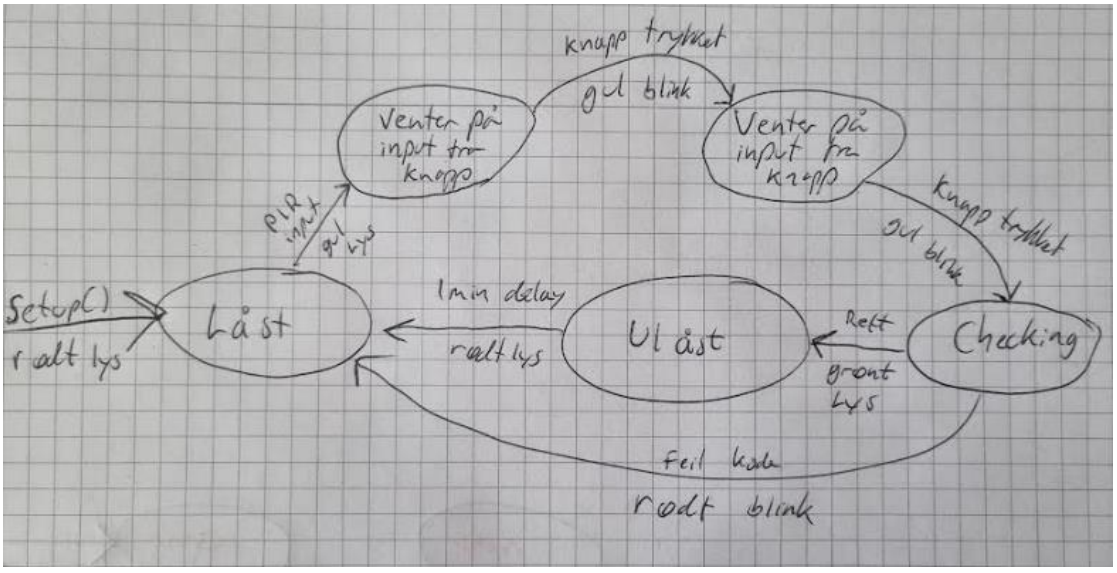


Figure 1: FSM, planlagt utgave

Som vist på bildet så ble denne tenkt løst ganske så likt som det som ble vist frem i forelesningen. Det var en fornuftig måte å gjøre det på, men underveis i kodingen viste det seg enklere å droppe den

ventende tilstanden da det oppstod litt problemer. Det vil si at i den endelige varianten blir sjekking av inntastet kode utført i *ventende* tilstand, og det er ikke en egen *sjekk*-tilstand.

Som FSM'en viser skal det røde lyset lys når den startes. Den inntar umiddelbart en låst tilstand, og når PIRen registrerer noe, er den klar til å ta imot input. Den går da inn i en ventende tilstand der den venter på både første trykk og andre trykk fra knappen. Når den står i en slik tilstand, skal det gule lyset lys. For hvert trykk på en knapp, blinker det gule lyset. De andre lysene er slukket i ventende tilstand. Etter å ha mottatt 2 trykk kjøres det en sjekk av input fra knappen. I den tilstanden er alle lys slukket, men i det den har sjekket vil maskinen sendes videre til en ny tilstand. Dersom korrekt kode, går den til ulåst-tilstand og grønt lys aktiveres. Her står den i et gitt antall sekunder, før den hopper til låst tilstand og det røde lyset aktiveres samtidig som det grønne slukkes. Dette er en sikkerhetsmekanisme som er viktig. Ved feil kode i sjekk-tilstanden, vil det røde lyset blinke og maskinen inntar låst tilstand igjen. Den ligger da og venter på nytt signal fra PIRen slik at man kan prøve på nytt.

Access Control Hardware/Software Implementation

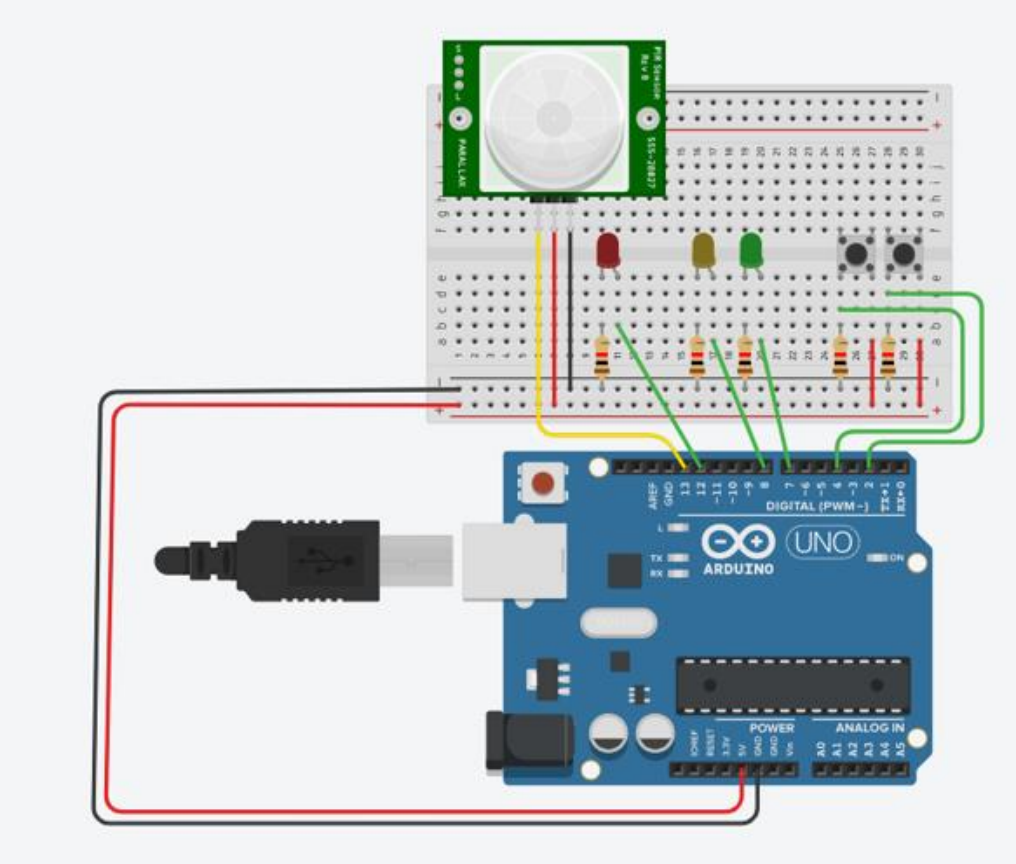


Figure 2: TinkerCAD krets

Bildet over viser TinkerCAD designet. 3 lys, 2 knapper, 1 PIR og et Arduino-brett, alle koblet sammen.

I denne delen av oppgaven overføres FSM-designet til kode. Dette ble løst med en setup()-metode der alt som er på brettet koblets til Arduino-enheten. Det vil si at de 2 knappene som er koblet til som input-punkt, mappes til riktig pin. Det samme gjøres for PIR-en sin input-del. De 3 lyskildene skal respondere på handlinger og fungere som en slags «feedback»-enhet. De er da output-kilder, og mappes deretter i koden. I tillegg settes alle sensorer til sine startverdier, det vil blant annet si at maskinens tilstand settes til 0. I systemet her betyr 0 at den er *låst*, mens 1 betyr *venter* og 2 betyr *åpen*. Dette er en kritisk del av koden.

Etter oppsettet går koden og maskinen inn i en evig loop. Ettersom den er satt til å ha tilstand 0 (låst) i setup, vil den treffe den delen av switch-setning som ligger som hovedelement i loop()-delen av koden.

Denne delen av koden er altså styrt av en switch-setning, som utfører kode basert på hvilken tilstand maskinen er i. Den har som beskrevet over 3 tilstander: *ulåst*, *venter*, *låst*. Avhengig av hvilken tilstand den havner i, vil det utføres det som er forventet av den gitte tilstanden.

Når den ligger i tilstand *låst*, som den automatisk vil gjøre ved oppstart, kreves det at signalet fra PIRen er høyt for at den skal skifte tilstand. Det naturlige er da å hoppe over til en *ventende* tilstand. Der kjøres det sjekk av input (hvilke knapp som trykkes på) og dette sammenlignes med en kode (som her er en egen variabel, for enkelhets skyld). I tillegg er der en teller som registrerer antall trykk på knappene. I det den blir større enn 2, vil maskinen hoppe tilbake til *låst* tilstand. Dette for å forenkle ting, samt sikre at riktig kode ikke kan komme etter f.eks ha tastet feil tall først.

Dersom knapp 1 og knapp 2 er trykker i riktig rekkefølge (matcher den lagrede koden), vil tilstanden hoppe til *ulåst* tilstand. Her står den et bestemt antall sekunder, før den automatisk går tilbake til *låst* tilstand.

Med hver enkelt tilstand følger det også endringer på lysene og hva de skal vise. *Ventende* har gule blink, korrekt kode og *åpen* tilstand har grønt, mens rødt naturligvis betyr *låst* og stengt. Dette er lagt inn i de forskjellige case-ene i switch-setningen.

REST API Cloud Service

Neste del av oppgaven var å implementere REST APIet og de nødvendige metodene via Spark-frameworket.

De består av følgende “routes”:

```
post("/accessdevice/log" …
get("/accessdevice/log" …
get("/accessdevice/log/:id" …
delete("/accessdevice/log" …
put("/accessdevice/code" …
get("/accessdevice/code" …
```

Disse er relativt selvbeskrivende. Den øverste lar oss registrere en rad i loggen. Denne loggen inneholder alle forsøk på innlogging, uavhengig av om det er vellykket eller ikke. Sammen resultatet av forsøk lagres

dette da som en «record» i loggen, og tildeles en ID. ID telleren starter på 0 når programmet starter. For hvert element («record») som legges til i loggen, vil IDen øke med 1.

Get-routen mot loggen henter ut hele loggen, men det er også mulighet til å hente ut ved å spesifisere ID til den «recorden» man ønsker å se.

Hele loggen kan tømmes med en delete-routing. Når den er kjørt vil det returneres en tom logg til brukeren.

For å bytte kode, kan en put-request sendes mot /accessdevice/code delen. JSON-dataen som sendes må da altså inneholde den nye koden. For å se koden, kjøres en enkel get-request mot samme område.

Selve loggen er lagret i en ConcurrentHashMap der nøkkelen er IDen til logg-recorden, og verdien er dataen den inneholder (i form av et AccessEntry-objekt), som da vil være f.eks meldingen «Access Denied».

Selve adganskoden lagres som et objekt av typen AccessCode. De inneholder en tabell med de to tallene som er satt som kode. Når appen startes opprettes det et nytt slik objekt, og i den er det allerede hardkodet inn at koden er 1,2.

I dette systemet er det mye frem og tilbake mellom JSON-objekter. Gson er hele tiden brukt til å gå både til og fra JSON-representasjon av dataen, avhengig av hva den gitte situasjonen og delen av programmet trenger. Ettersom dataen som sendes via http-requesten er i JSON-formatet, må dette gjøres på begge sider av systemet.

En egen to-Json()-metode er da en naturlig hjelper i dette prosjektet. Denne bygger videre på Gson sine innebygde metoder.

Device Communication

I selve REST-klienten er det 2 viktige metoder: doPostAccessEntry og doGetAccessCode. Her valgte jeg å benytte meg av biblioteket som heter OkHttp – som foreslått i oppgaveteksten.

I metoden doPostAccessEntry() skal registreres en hendelse i loggen. Denne kalles dersom det er et mislykket forsøk på å logge inn (krever at maskinen er i en ventende tilstand), eller når det i samme ventende tilstand er et vellykket forsøk på innlogging. Som del av meldingen skrives også hva som skjer, det vil si om at det gitt tilgang eller ikke, samt hvilken tilstand maskinen står i på tidspunktet meldingen skrives til loggen.

Metoden er bygget opp med OkHttp. Det lages en OkHttp-klient, det spesifiser hvilken type media som kommer, og med denne informasjonen bygges det en request. Via den tidligere brukte Gson-biblioteket lages det en JSON-representasjon av melding vi ønsker å plasser i loggen. Denne melding vil være body-delen av http-requesten som etterpå bygges opp og settes sammen.
Det hele pakkes til slutt inn i response-meldingen.

Metoden doGetAccessCode() har som jobb å hente ut den nåværende tilgangskoden. IoT-enheten vil hele tiden trenge denne for å kunne sjekke om inntastet kode er riktig eller ikke.

Også her brukes OkHttp-løsningen til å bygge en request og en response-melding. Sammen med dette gjøres det også den nødvendige «konverteringen» til og fra JSON, via Gson.

System Testing

Testingen av systemet ble utført ved å bruke Postman og den grafiske fremstillingen av IoT-enheten som ble levert med oppgaven.

I del A ble testene utført direkte i TinkerCAD. I tillegg ble det også under testing brukt debuggeren der, men dette er ikke med i den endelige koden.

Postman viste seg å være et meget nyttig verktøy til å sjekke at REST-servicen er oppe og kjører, og at alle GET, POST, PUT og DELETE meldinger sendes korrekt og gjør det de skal. Dette var flittig brukt for å verifisere at hver enkelt implementasjon i RestClient.java ble korrekt.

Den endelige testen ble utført ved å først starte appen tilknyttet REST-servicen, kjøre den virtuelle IoT-enheten og knytte den til nettverket via N-knappen, og så fyre opp Postman ved siden av dette. Deretter brukes knappene til å teste koder og innlogging i IoT-enheten. Etter hver test der ble det kjørt søk mot loggen via en GET-request i Postman, for å verifisere at ting var korrekt. Dette var også synlig i terminalen i Eclipse. Alle metodene ble teste på denne måten.

For full oversikt over testene, se dokumentet med screenshots. Det finnes her:

https://github.com/h131203/dat110-project4-startcode-iotdevice/blob/master/Prosjektrapport_v1%20-%20screenshots.pdf

Konklusjon

Dette prosjektet gikk relativt greit. Det var lærerikt og ganske så nyttig å få en oppgave der resultatet av ting man gjør er veldig synlig med en gang, slik som f.eks TinkerCAD og brettet der.
Jeg fikk løst alt, men noe er kanskje litt knotete løst.

Koden til Arduino-brettet og den tenkte FSM-en ble en større utfordring enn planlagt, og jeg endte opp med en litt annerledes løsning enn planen var. I tillegg kunne koden her vært skrevet mye penere og bedre (egne metoder for å styre lys og resette tellere stod på planen). Jeg opplevde litt problemer med å få lysene til å lyse når det skulle, men etter å ha satt opp brettet på nytt flere ganger (med samme oppsett?!) lyst det plutselig.

Spark-rammeverket var nytt for meg, og jeg brukte litt tid på å koble dette korrekt mot adressene i skyen. Det ble en del feilmeldinger av typen «*The requested route … has not been mapped in Spark for Accept*». Men sånt lærer man av.

Oppsummert vil jeg si dette var et prosjekt som gikk fint og hadde gode utfordringer, samtidig som det viste potensialet og mulighetene til IoT-er på en fin måte.