

|  |
| --- |
| DTU Diplom – 13. juni 2017 |
| Data samling |
| Bachelorprojekt 2017 |
|  |

**Vejledere:** Henrik Tange & Ole Schultz.

**Studerende:**

|  |  |
| --- | --- |
| Nicklas Ejberg Storm Jensen - s135274 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Abdirazak Mohamud Yusuf - S130599 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Hakan Yurdakul Pedersen - S133835 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Indhold

[2 Forord 6](#_Toc485114591)

[3 Introduktion 6](#_Toc485114592)

[3.1 Baggrund 6](#_Toc485114593)

[3.2 Virksomheds introduktion 6](#_Toc485114594)

[3.3 Formål 6](#_Toc485114595)

[3.4 Metode 7](#_Toc485114596)

[4 Problemformulering 7](#_Toc485114597)

[4.1 Krav 7](#_Toc485114598)

[4.2 Servere 8](#_Toc485114599)

[4.3 Indhentning af data 8](#_Toc485114600)

[4.4 Opbevaring af data 8](#_Toc485114601)

[4.5 Front-end 9](#_Toc485114602)

[4.6 Databehandling 9](#_Toc485114603)

[4.7 Milestone plan 10](#_Toc485114604)

[5 Problemanalyse 12](#_Toc485114605)

[5.1 Server, Mono og .NET 12](#_Toc485114606)

[5.1.1 Web-server 12](#_Toc485114607)

[5.1.2 Mono 14](#_Toc485114608)

[5.1.3 .NET 14](#_Toc485114609)

[5.2 Indhentning af data 15](#_Toc485114610)

[5.3 Opbevaring af data 16](#_Toc485114611)

[5.4 API 17](#_Toc485114612)

[5.4.1 Token 18](#_Toc485114613)

[5.5 Front-end 19](#_Toc485114614)

[5.5.1 System type 19](#_Toc485114615)

[5.5.2 Programmeringssprog 20](#_Toc485114616)

[5.5.3 Visualisering 21](#_Toc485114617)

[5.5.4 Sikkerhed 21](#_Toc485114618)

[5.6 Databehandling 22](#_Toc485114619)

[5.6.1 Zeppelin 22](#_Toc485114620)

[5.6.2 Jupyter 23](#_Toc485114621)

[5.6.3 Vores egen platform 24](#_Toc485114622)

[6 Projekt afgrænsning og metoder 26](#_Toc485114623)

[6.1 Risikoanalyse 26](#_Toc485114624)

[6.2 Use case diagram 27](#_Toc485114625)

[6.3 Brugsscenarier 28](#_Toc485114626)

[6.4 Kravsporing 32](#_Toc485114627)

[7 Problemløsning 33](#_Toc485114628)

[7.1 Server 33](#_Toc485114629)

[7.1.1 Nginx opsætning 34](#_Toc485114630)

[7.1.2 Database firewall 35](#_Toc485114631)

[7.1.3 SSL certifikat 35](#_Toc485114632)

[7.2 Indhentning af data 36](#_Toc485114633)

[7.2.1 Design 36](#_Toc485114634)

[7.2.2 Implementering 36](#_Toc485114635)

[7.3 Opbevaring af data 43](#_Toc485114636)

[7.3.1 ER-Diagram 43](#_Toc485114637)

[7.3.2 RID 43](#_Toc485114638)

[7.4 API 43](#_Toc485114639)

[7.4.1 Implementering 44](#_Toc485114640)

[7.4.2 Swagger 49](#_Toc485114641)

[7.4.3 Token 50](#_Toc485114642)

[7.5 Front-end 51](#_Toc485114643)

[7.5.1 Systemtype 51](#_Toc485114644)

[7.5.2 Programmeringssprog 52](#_Toc485114645)

[7.5.3 Visualisering 53](#_Toc485114646)

[7.5.4 Sikkerhed 54](#_Toc485114647)

[7.6 databehandling 55](#_Toc485114648)

[8 Test 59](#_Toc485114649)

[8.1 Server 59](#_Toc485114650)

[8.2 TimerJob 60](#_Toc485114651)

[8.3 API 64](#_Toc485114652)

[8.4 Front-end 67](#_Toc485114653)

[8.4.1 Acceptance test 67](#_Toc485114654)

[8.4.2 Unit test 69](#_Toc485114655)

[9 Konklusion 73](#_Toc485114656)

[9.1 Produktvurdering 73](#_Toc485114657)

[9.2 Procesvurdering 76](#_Toc485114658)

[9.3 Forbedringer 77](#_Toc485114659)

[10 Appendiks 78](#_Toc485114660)

[10.1 Database ER-diagram 78](#_Toc485114661)

[10.2 RID 80](#_Toc485114662)

[10.3 API klassediagram 80](#_Toc485114663)

[10.4 Front-end klassediagram 85](#_Toc485114664)

[10.5 Forsøg på egen platform 86](#_Toc485114665)

[10.5.1 Delkonklusion 87](#_Toc485114666)

Figuroversigt

[Figur 1: Nginx event processer 13](#_Toc485114667)

[Figur 2: Illustrering af hvordan Mono er opbygget 14](#_Toc485114668)

[Figur 3: Indhentning data fra sensor til database 15](#_Toc485114669)

[Figur 4: Applikationens brug af JWT 18](#_Toc485114670)

[Figur 5: Udsnit af token 19](#_Toc485114671)

[Figur 6: Zeppelin Architecture 23](#_Toc485114672)

[Figur 7: Jupyter dashboard 24](#_Toc485114673)

[Figur 8: Illustrere Spark core 25](#_Toc485114674)

[Figur 9: Usecase diagram 27](#_Toc485114675)

[Figur 10: Arkitektur diagram 33](#_Toc485114676)

[Figur 11: Site-avaliable/default 34](#_Toc485114677)

[Figur 12: Firewall 35](#_Toc485114678)

[Figur 13: Flowchart for IC-Meter timerjob 37](#_Toc485114679)

[Figur 14: IC-Meter data i JSON format 38](#_Toc485114680)

[Figur 15: IC-Meter sensor data tabel 39](#_Toc485114681)

[Figur 16: IC-Meter sensor tabel 39](#_Toc485114682)

[Figur 17: Flowchart for ThingSpeak timerjob 40](#_Toc485114683)

[Figur 18: ThingSpeak data i JSON format 41](#_Toc485114684)

[Figur 19: Firebase data i JSON format 42](#_Toc485114685)

[Figur 20: Opret IC-Meter sensor fra hjemmesiden 44](#_Toc485114686)

[Figur 21: IC-Meter API metoder 45](#_Toc485114687)

[Figur 22: Opret ThingSpeak sensor fra hjemmesiden 46](#_Toc485114688)

[Figur 23: ThingSpeak API metoder 46](#_Toc485114689)

[Figur 24: Opret custom sensor fra hjemmesiden 47](#_Toc485114690)

[Figur 25: LINQ’s benyttelse for at gemme sensoren i databasen. 47](#_Toc485114691)

[Figur 26: Illustrere hvad tabellerne indeholder 48](#_Toc485114692)

[Figur 27: Gateway API metoder 49](#_Toc485114693)

[Figur 28: Metode vist i swagger 50](#_Toc485114694)

[Figur 29: Arkitektur diagram 51](#_Toc485114695)

[Figur 30: MVC struktur 52](#_Toc485114696)

[Figur 31: Graf over CO2 53](#_Toc485114697)

[Figur 32: Database udklip af en bruger 54](#_Toc485114698)

[Figur 33: Binær værdi af password 54](#_Toc485114699)

[Figur 34: Tekst værdi af password 54](#_Toc485114700)

[Figur 35: Hardware minimumskrav 56](#_Toc485114701)

[Figur 36: Zeppelin installationsfejl 57](#_Toc485114702)

[Figur 37: Miljø minimumskrav 58](#_Toc485114703)

[Figur 38: Destination for API 59](#_Toc485114704)

[Figur 39: Destination for front-end 59](#_Toc485114705)

[Figur 40: Kommando for at køre et dotnet program 59](#_Toc485114706)

[Figur 41: Udklip af log fra dotnet program 59](#_Toc485114707)

[Figur 42: IC-Meter sensor tabellen 60](#_Toc485114708)

[Figur 43: IC-Meter sensor processering 61](#_Toc485114709)

[Figur 44: IC-Meter values tabellen 62](#_Toc485114710)

[Figur 45: Udklip af topbaren fra hjemmesiden 62](#_Toc485114711)

[Figur 46: Udklip af login siden på Zeppelin 63](#_Toc485114712)

[Figur 47: Udklip af Zeppelin notebook 63](#_Toc485114713)

[Figur 48: Tilføj custom sensor - udfyldt 64](#_Toc485114714)

[Figur 49: Custom JSON data 65](#_Toc485114715)

[Figur 50: Udsnit af koden, der tilføjer JSON til databasen 65](#_Toc485114716)

[Figur 51: SQL-query til custom sensor 65](#_Toc485114717)

[Figur 52: Datasæt for SQL-query 66](#_Toc485114718)

[Figur 53: Graf for custom sensor 66](#_Toc485114719)

[Figur 54: Loginsiden 68](#_Toc485114720)

[Figur 55: Tilføj sensor siden 69](#_Toc485114721)

[Figur 56: Test af Index() 69](#_Toc485114722)

[Figur 57: Test af DataSouces() 70](#_Toc485114723)

[Figur 58: Test af Data med IC-Meter som input 70](#_Toc485114724)

[Figur 59: Test af Data() med ThingSpeak og Custom som input 71](#_Toc485114725)

[Figur 60: Illustration af hvordan det ser ud på hjemmesiden 86](file:///C:\Users\Nicklas\Google%20Drev\7.%20semester\Dokumentation\Samlede%20rapport\Rapport%20afgangsprojekt%203.0.docx#_Toc485114726)

# Forord

Denne rapport er rettet mod det tekniske personale, der skal vedligeholde eller viderebygge på dette system.  
Rapporten indeholder en beskrivelse af implementeringen af systemet, beskrivelse af udførte tests samt overvejelser omkring design og implementering ud fra de angivne krav.  
Rapporten er blevet til over en periode på ca. 4 måneder, hvor den er udfærdiget sideløbende med at projektet er blevet udviklet.

# Introduktion

## Baggrund

I dag er der meget fokus på CO2 udslippet verden over, og derfor skal der lægges fokus på forbrugeren. Da forbrugerne ikke altid tænker lige meget over hvor meget vand, varme, strøm m.m. de bruger, skal der kastes lys på det over for forbrugeren. Dette kan gøres ved at publicere forbruget af alle ressourcer, så hver enkelt forbruger kan se hvor meget der bliver brugt, f.eks. hos DTU Campus Ballerup, hjemmet eller en boligforening.

Udover at visualisere ressourceforbruget, har vi i dag decentrale sensorer for el, varme, vand og gas, der kan aflæses, men det kræver man selv aflæser sensorerne. Dette betyder at data skal samles et sted så forbrugeren har et overblik over alt forbrug.

## Virksomheds introduktion

Virksomheden for dette projekt er DTU. Repræsentant og projektleder for DTU er Ole Schultz. Han er lektor og underviser ved DTU, og er manden der står bag ideen om data aggregeringsplatformen, som vi udvikler. Ole Schultz vil derfor agere som leder for projektet, men også som vejleder for os. Ud over dette, er han en af mange, som kommer til at bruge det færdige produkt.

## Formål

Der skal bygges en IT platform, som skal kunne håndtere store mængder data, der kommer fra mange forskellige datakilder. Systemet skal kunne håndtere, at dataene kommer asynkront, samt der kan komme store mængder på nogle tidspunkter, og mindre mængder på andre tidspunkter. Mellem applikationslageret og serverlageret skal der håndteres server belastning, så systemet ikke har let ved at bryde sammen. Systemet skal lagre denne data i en database, og kunne fremvise dataene til brugerne grafisk, samt analysere på dataene. Yderligere skal der undersøges hvorvidt systemet er skalerbart, og hvordan der kan registreres nye sensorer i fremtiden.

## Metode

Projektet bliver udviklet agilt, dvs. at vi hver uge holder møde med vores projektleder Ole Schultz. Ved disse møder drøfter vi status på projektet, hvad der er lavet, samt hvad der skal laves til næste møde. Ole Schultz kommer med nye ideer til projektet, og da vi udvikler agilt, prøver vi så vidt muligt at implementere de nye ideer, eller forbedringsforslag han har. Det betyder at vi løbende vil ændre vores prioritering af hvad der skal laves først, sådan at vi bedst muligt kan tilfredsstille slutbrugeren. Det betyder også at vi ikke følger den klassiske vandfaldsmodel, hvor man går fra fase til fase sekventielt, og designer hele systemet fra starten. Vi vil løbende ændre i kravene, og designet, altså hoppe frem og tilbage i faserne, for at tilpasse os kravene fra Ole Schultz og DTU.   
Grunden til vi har valgt denne udviklingsmetode er, at vi har mulighed for at have tæt kontakt til Ole Schultz, da han er villig til at være meget involveret i projektet. Derudover mener vi at man får et bedre produkt, når man har tæt kontakt til slutbrugerne, og ”kunderne”. Kravene til systemet og designet vidste vi fra starten ville ændre sig løbende, da der var mange ukendte faktorer, heriblandt hvilke ressourcer vi havde tilgængelig, som f.eks. servere og sensorer, og pga. dette egner agil udvikling sig bedre til dette projekt.

# Problemformulering

## Krav

Efter at have snakket med Ole Schultz om hvad hans forventninger er til projektet, er vi kommet frem til følgende krav, som det endelige produkt skal opfylde.

Must have:

1. Alt data skal ligge i databasen, uden at det på noget tidspunkt vil blive slettet.
2. Webløsningen skal kunne tilgås fra alle platforme.
3. Webløsningen skal illustrere dataene på en spændende måde.
4. Webløsningen skal muliggøre at brugere kan sammenligne data.
5. Det skal være muligt at kunne hente dataene i flere formater fra webløsningen.
6. Timerjob skal kunne hente dataene periodisk fra ThingSpeak og IC-meter.
7. Systemet skal kunne håndtere variable indkomne datamængder.
8. Systemet skal kunne håndtere excel-data.
9. Systemet skal indeholde en administrations side, hvor der kan håndteres autorisering.
10. Systemet skal kunne håndtere variation af belastning.
11. Der skal analyseres hvorvidt systemet kan skaleres.
12. Det skal være muligt at registrere nye sensorer til system via en gateway.

Nice to have:

1. Android app

## Servere

Da det er et fastlagt krav, at det skal være muligt at gøre front-end tilgængelig på nettet, kræves det at front-end køre på en web-server. Da web-serveren er adskilt fra databaseserveren, skal der være en sikker forbindelse mellem disse to. Web-serveren skal derimod være offentlig tilgængelig, så alle brugere kan tilgå den. I denne forbindelse er der behov for at finde ud af hvilken web-server vi skal benytte, samt hvordan vi kan få systemet til at køre på serverne.

## Indhentning af data

Da mange af de eksisterende sensorer på skolen sender data til f.eks. IC-Meter og ThingSpeak, skal vi udvikle et modul som kan hente data fra disse, og gemme dem I vores egen database. IC-Meter og ThingSpeak er selvstændige firmaer, som opbevare data for kunder, og derefter udstiller en web-side og et web API, hvor kunderne kan se deres data. Vi skal finde ud af hvordan disse API’er fungerer, og hvordan vi skal bruge dem til at hente data. Da de fleste sensorer sender data flere gange I timen, skal vi finde ud af hvor ofte vi skal hente data. Vi kan ikke regne med at disse API’er er fejlfrie, og vi skal derfor finde ud af hvordan vi håndtere f.eks., at et API går ned, eller vi intet svar får.

## Opbevaring af data

Da vi skal opbevare alt data vi henter og modtager i vores egen database, er der et par ting, som er vigtige at vi tænker på i dette henseende. Vi skal være opmærksomme på hvor meget en sensor måling fylder i databasen i forhold til hvor mange målinger vi henter. Derudover skal vi finde ud af hvordan vi smartest kan opbevare dataene, både for at det fylder mindst muligt, men også så vi nemt og hurtigt kan hente det igen.  Vi skal også tænke over hvilket DBMS (Database Management System) vi skal bruge, da der er forskel i hvordan det gemmer og behandler data.

## Front-end

De data, der bliver hentet ind fra andre firmaer, samt de data der bliver sendt til os fra andre sensorer, skal på en pæn visuel måde fremstilles for brugeren. Der skal diskuteres om det skal være en hjemmeside, hvor alle kan klikke ind og se dataene, uden at installere noget på deres lokale maskine, eller om de skal installere et program på computeren, hvorfra de kan få visualiseret dataene. Samtidig skal der ses på hvordan der på den pæneste mulige måde kan visualiseres data. Da alt data ikke skal være let tilgængeligt for alle, skal der diskuteres hvilken sikkerhed der skal bruges, samt hvilke data der ikke skal være tilgængelige for alle.

## Databehandling

Ud over en visualisering af data, skal der også være mulighed for at arbejde med det. Det betyder at der skal være mulighed for at vælge en eller flere grafer, placere dem i et diagram, hvor efter der kan ses på forskelle og ligheder mellem dem. Dette vil kunne give en forbruger mulighed for at se om de f.eks. har brugt mere strøm året eller måneden forinden, og ud fra andre data se hvad der er gjort for at få denne ændring, dette kunne f.eks. være hvis der har været flere mennesker til at bruge strømmen en måned.

Samtidig skal data kunne ligges sammen eller trækkes fra hinanden, så man ender ud med f.eks. maksimum temperatur for en måned eller et år, eller den laveste strømforbrugsperiode i løbet af året. Dette vil give forbrugeren muligheder for at se hvad der er gået galt, siden der f.eks. er brugt mere strøm i en periode, eller at temperaturen i nogle værelser er højere end nødvendigt. I sidste ende vil dette kunne spare forbrugeren for unødigt forbrug af ressourcer.

## Milestone plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Milestone | Status når udført | Ansvar | Udført | Projektleverance |
| Milestone 1 | | | |  |
| Milestoneplan | Færdig | Alle | X | **I milestone 1 opnås:** |
| Projektbeskrivelse | Færdig | Hakan | X | Projektets omfang er analyseret |
| Kravspecifikation | Færdig | Alle | X |
| Kravsporingsmatrix | Færdig | Hakan | X |
| Risikoanalyse |  | Alle | X |
| Brugsscenarier | Færdig | Alle | X |
| Start på rapport | Udkast | Hakan | X |
| Opsætning af server | Færdig | Abdi | X |
| Opstart af database | Opstart | Hakan | X |
| ER-Diagram | Udkast | Hakan / Nicklas | X |
| Analyse af datamængder | Færdig | Abdi / Hakan | X |
| Opstart af hjemmeside | Opstart | Nicklas | X |
| Milestone 2 | | | |  |
| Timerjob henter data til databasen | Opstart | Hakan | X | **I milestone 2 opnås:** |
| API kan hente data fra database | Opstart | Abdi | X | Systemet begynder at kunne basisfunktioner |
| Remote forbindelse til database | Færdig | Alle | X |
| Problemanalyse | Færdig | Alle | X |
| Hjemmeside kan basis dataanalyse | Færdig | Nicklas | X |
| Database | Færdig | Hakan | X |
| Problemformulering | Færdig | Alle | X |
| Milestone 3 | | | |  |
| Implementering af gateway | Færdig | Nicklas | X | **I milestone 3 opnås:** |
| API håndterer nye sensorer | Færdig | Abdi | X | Systemet fungerer og mangler enkelte funktioner |
| Timerjob håndtere nye sensorer | Færdig | Hakan | X |
| Database tilretning | Færdig | Hakan | X |
| Milestone 4 | | | |  |
| Gateway på hjemmeside | Færdig | Nicklas | X | **I milestone 4 opnås:** |
| API | Færdig | Abdi | X | System er fuldt fungerende |
| Timerjob | Færdig | Hakan | X |
| Test af systemet | Opstart | Alle | X |
| Milestone 5 | | | | 13-06-2017 |
| Dokumentation - Test | Færdig | Alle | X | **I milestone 5 opnås:** |
| Dokumentation - Forord | Færdig | Nicklas | X |  |
| Dokumentation - Problemformulering | Færdig | Alle | X |
| Dokumentation - Problemanalyse | Færdig | Alle | X |
| Dokumentation - Løsningsstrategi | Færdig | Hakan | X |
| Dokumentation - Problemløsning | Færdig | Alle | X |
| Dokumentation - Diverse til appendiks | Færdig | Alle | X |
| Dokumentation - Produktvurdering | Færdig | Alle | X |
| Dokumentation - Procesvurdering | Færdig | Alle | X |
|  |  |  |  |
| Dokumentation - Sammensætning af rapport | Færdig | Alle | X |

# Problemanalyse

## Server, Mono og .NET

* **Windows Server**

Windows server er en gruppe af operative systemer, som er designet af Microsoft. En af fordelene ved en Windows server er, at deres hovedformål er GUI (Graphical User Interface), der for nogen kan gøre det mere overskueligt at finde rundt i. Fordele ved at benytte en Windows server er at installationen for blandt andet Asp.net og API vil være lige til, da det er muligt at installere programmet direkte i gui’en. Ulemperne er at en Windows server er licens baseret, som gør, at man skal betale for at bruge det. Samtidig er der ikke mulighed for at rette op på fejl eller mangler, da en Windows server ikke er open-source.

* **Linux Server**

Linux server benytter sig oftere af en terminal i stedet for en brugergrænseflade. Dette kan virke mere overskueligt for andre. I forhold til stabilitet er det i Linux muligt og ændre configurations filerne uden at genstarte Linux serveren, hvorimod i Windows skal du genstarte serveren. Da Linux serveren er open-source, giver dette muligheder for at kunne implementere ting selv, som ikke er blevet implementeret før, samt at rette fejl selv, som man ikke har tid til at indrapportere og vente på bliver ordnet. Derudover koster det intet at have en Linux server.

### Web-server

I kravene er det angivet, at der skal benyttes en front-end til at vise dataene grafisk, og at der skal være et API, hvor man ved hjælp af metodekald kan hente eller gemme data. Dette kræver at man har en web-server, hvorpå hjemmesiden og API’et kan være installeret. Der findes mange slags web-server software. Vi har valgt at analysere følgende: Apache og Nginx, da de er nogle af de mest benyttede.

Web-server er et program som benytter HTTPS (HyperText Transfer Protocol Secure) eller HTTP (HyperText Transfer Protocol), til at levere website eller filer. Hver web-server har et domænenavn og en IP-adresse. Ved indtastning af en URL, for eksempel http://www.google.com i browseren, vil en forspørgelse blive sendt til serveren, der vil behandle forespørgslen og sende hjemmesidens indhold retur. Enhver computer kan bruges som en web-server ved at installere et web-serverprogram på maskinen, så længe den har internetadgang. Eksempler på web-serverprogrammer er Apache, IIS og Nginx. Der findes andre såsom Novelles Netware-server og google.

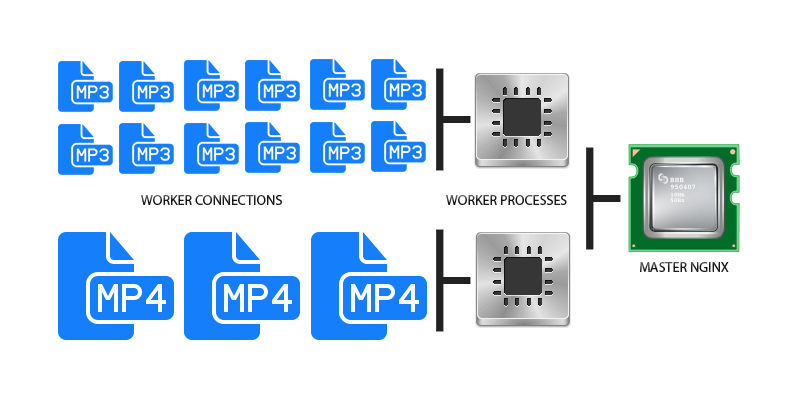
* **Apache**

Apache server er en web-server, som er en solid platform for en hvilken som helst web teknologi. Apache er en open-source web-server applikation, som kan køre på alle større operativsystemer som (Linux, Windows, Unix, OSX). Apache er kendt for at være anvendt til Linux. Hvis man kombinerer en Linux server med Apache, MySQL-database og PHP scripting, bliver det til den populær LAMP web-server løsning.

* **Nginx**

Nginx er en open-source HTTP web-server og reverse proxy server. En proxy server er noget som optræder i stedet for internettet, det vil sige at det er bindeleddet mellem den lokale computer og internettet. Den bliver benyttet til at gemme kopi af data, så de kan hentes hurtigere når den bliver genindlæst. Reverse proxy server er det som ligger bag firewallen i et privat internet og som dirigerer klients forespørgsler til den rette backend server. Den kan yderligere også kontroller flowet af netværket mellem klienten og server.

Nginx følger event-baseret proces, hvilket vil sige at, en masse events eller aktiviteter bliver placeret i en proces “container”. Når en bruger har brug for en event eller aktivitet, bliver den specifikke fundet frem. Dette kan ses på Figur 1.

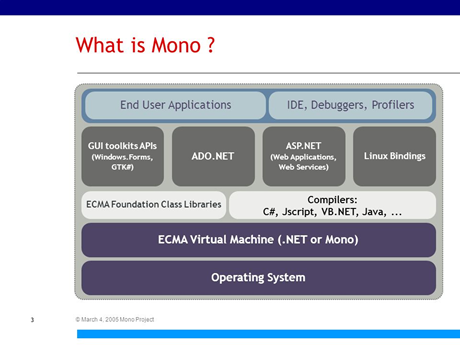


Figur 1: Nginx event processer

En af fordelene, som Nginx har er at den er hurtigere når den anvendes til statiske sider, da den benytter mindre hukommelse, hvilket skyldes brugen af en single thread. For at kunne kører ASP.Net Core projektet kræves der et modul eller en oversætter som kan håndterer dotnet på en Linux server.

### Mono

Mono er et open-source modul for .Net udviklere som et lavet ud fra ECMA standard (*European Computer Manufacturers Assoiation)* for C# og Common Language Runtime. Mono er ikke et programmeringssprog som Java, Python eller PHP, men en CLI oversættelse. Desuden er den designet som en multiplatformsapplikation, der kan kører på Linux.



Figur 2: Illustrering af hvordan Mono er opbygget[[1]](#footnote-1)

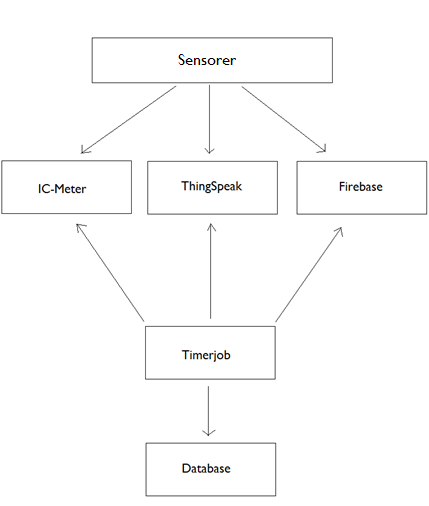
### .NET

.NET CLI er en ny cross platform til at udvikle .NET Core applikationer. Der findes to typer .NET Core applikationer: Self-Contained deployment og Framework-Dependent deployment.

Self-contained deployments betyder at man ikke kun deployer projektet og alle dens afhængigheder, men også versionen af .Net core. Desuden er du garanteret at projektet kan køres et andet sted med et USB stik. Den anden applikation ”Framework- Dependent deployment”, som er anderledes i forhold til Self-contained deployment. Her indeholder den ikke rester af .Net bibliotekerne men kun koden og dens tredjeparts afhængigheder.

## Indhentning af data

Det data som skal lægges i vores database kommer fra forskellige steder. Noget bliver sendt til os via vores API, da nye sensorer kan sættes op til dette. Andre sensorer sender til andre firmaer, som IC-meter eller ThingSpeak, som også kan illustrere data i en visuel form. Vi skal hente data fra firmaerne med passende interval. For at dette kan lade sig gøre skal vi udvikle et program som henter disse data. Ideen er illustreret i Figur 3:



Figur 3: Indhentning data fra sensor til database

Når vi skal programmere programmet, har vi forskellige muligheder. Vi kan lave programmet så det henter dataene sekventielt, dvs. at den eksempelvis først henter fra IC-Meter, derefter ThingSpeak og til sidst Firebase. Dette vil fungere, men programmets skalerbarhed er ikke optimal. Hvis vi i stedet får programmet til at køre i forskellige processer, som hver især henter data fra firmaerne parallelt, vil systemet være mere skalerbart. Hvis vi siger, at programmet skal hente data hvert 5 minut fra hvert sted, og der er 1000 sensorer hvert sted, er det ikke sikkert den kan nå at hente alle 3000 sensorer på under 5 minutter, og programmet vil derfor komme bagud med at hente målingerne. Hvis det derimod henter data fra de 3 steder parallelt med hinanden, vil det være meget mere skalerbart.

Timerjobbet skal køre 24 timer i døgnet, da sensorer også sender 24 timer i døgnet, men vi skal have fundet ud af hvor ofte programmet skal hente data fra firmaerne. Sensorerne sender data med forskellige intervaller, nogle sender flere gange i minuttet, og andre sender et par gange i timen. Vi kan vælge at gemme et sted i databasen, hvor ofte sensorerne sender, og hente data i forhold til dette. På denne måde vil vi have de nyeste data hele tiden. Vi vil oftere skulle hente data, da nogle sensorer sender flere gange i minuttet. Dette vil være mindre skalerbart.

Det virker smartere at hardcode en værdi i applikationen, så den henter data f.eks. hvert 5 minut. Dette vil sikre at vi altid har de nyeste data, og det vil gøre applikationen mere simpel.

Det er også vigtigt at vi sikrer, at programmet kan håndtere, hvis den ikke får noget svar fra et firmas API. Det kunne ske at deres API gik ned i et tidsrum, og her er det vigtigt programmet ikke fejler. Denne værdi skal være mulig for brugeren at rette, f.eks på hjemmesiden.

En anden men meget vigtig ting med timerjobbet er sikkerhed. Nogle sider kræver at man sender et brugernavn og kodeord for at få adgang til sensordataene. Det er meget vigtigt, at brugernavne og koder som timerjobbet henter fra databasen er krypteret, og at timerjobbet kan dekryptere dem og derefter sende dem via en sikker protokol til firmaernes API. Derfor skal der implementeres dekryptering i applikationen.

## Opbevaring af data

Da vi skal opbevare store mængder data, er det vigtigt at vi designer en smart database, der er normaliseret, og gerne har indeks på de kolonner som vi med stor sandsynlighed kommer til at søge oftest i. Samtidig skal vi tage højde for rettigheder for de enkelte brugere, så der kan holdes styr på hvilke brugere der adgang til hvilke sensorer. der skal defineres om brugere har adgang til at slette, tilføje eller rette i de forskellige tabeller. Vi skal også vælge den DBMS som passer bedst til vores system. I forhold til dette, er der et par muligheder vi kan vælge imellem.

* **MYSQL**

Fordelen ved at vælge MYSQL er at det er den DBMS vi kender bedst. Det er en database vi har brugt i andre projekter, og er et stabilt valg, da vi har gode erfaringer med det. Vi har sat en MYSQL database op succesfuldt flere gange før, og kender til opsætningen og indstillingerne af databasen. Alt i alt kan den det vigtigste ting en DBMS skal kunne, og ville være en fint valg.

* **POSTGRESQL**

Postgresql er ikke noget nogen af os har arbejdet med før, men det er noget vi har hørt om flere gange. Der er flere funktioner indbygget i dette i forhold til MYSQL. Det kunne være brugbart med ekstra funktioner hvis vi skal lave avancerede søgninger i vores database. En ting vi skal tage med i vores beslutning om DBMS er at ingen af os kender til postgresql platformen, og der vil derfor gå ekstra tid med at komme ind i systemet.

* **NOSQL**

En tredje mulighed er, at vi vælger at lave vores database som en ”NOSQL” database, eksempelvis MongoDB eller CouchDB. Dette er noget ingen af os har arbejdet med, og vil tage tid at forstå og komme i gang med, men det er bestemt en mulighed da det er designet med henblik på big data. NOSQL databaser har bedre ”performance” end eksempelvis MYSQL når det kommer til big data, men har færre indbygget funktioner. Eksempelvis er constraints og joins ikke indbygget i systemet. Da det kan forekomme at sensorerne har fejl i en måling, er det vigtigt vi har en form for constraint i vores database. NOSQL kan dog indeholde forskellige typer data som kunne være brugbart for os. Man skal ikke på forhånd definere antallet af kolonner, hvilke datatyper kolonnerne er osv., men det kan også være, at det er bedre for os at have en struktureret database, for at sikre at vi bruger mindst plads, og har den mest forståelige struktur. Dette gør det nemmere for fremtidige udviklere at overtage projektet og videreudvikle på det.

## API

Et API er applikation som bruges som mellemled mellem en database og en klient. API’et skal håndtere logikken på den måde, at en klient vil sende en forspørgelse til API’et. Det kan f.eks. være, at klienten ville have noget bestemt data fra databasen eller at en sensor vil sende data til databasen. API’et vil finde det nødvendige data i databasen og sende det retur til klienten. Formålet med API’et er at der skal stilles nogle funktioner til rådighed, som klient applikationer kan tilgå. Dette vil gøre det muligt at tilføje f.eks. en app til projektet, uden de store ændringer.

Der findes flere typer af API’er, som f.eks. SOAP og REST.

* **SOAP**

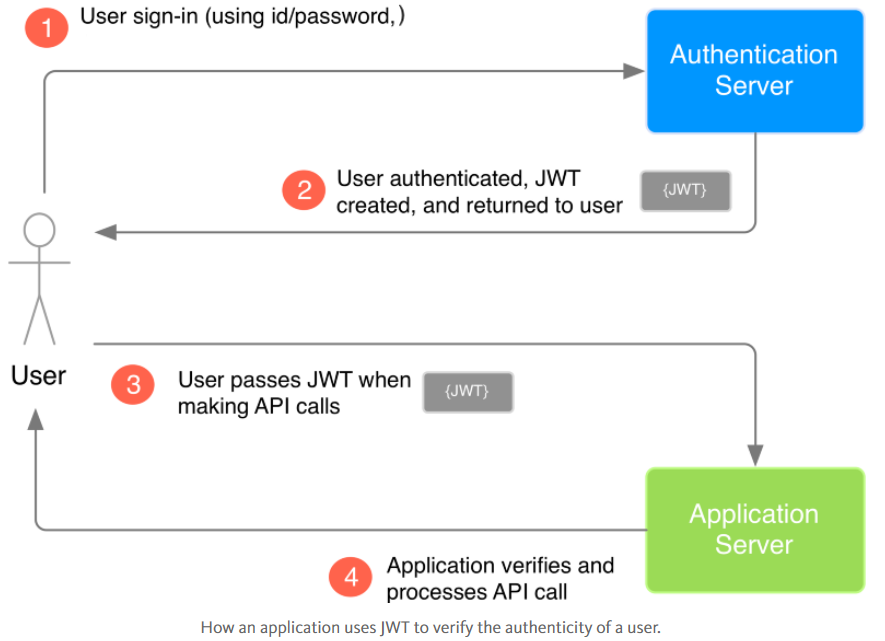
SOAP er en protokol, der sender XML-baserede beskeder, og udveksler information mellem computere. SOAP yder data transport for web-services. SOAP gør det muligt for klient applikationer at kalde de metoder API’et indeholder. Strukturen for SOAP er en *envelop*, som indeholder en header og en body. Headeren indeholder data om hvor *envelopen* skal sendes til, mens body indeholder det data der sendes.

* **REST**

REST står for Representational State Transfer. Det er et web service, der benytter HTTP. Det fungerer ligesom en hjemmeside i en browser, dog er den Stateless, hvilket betyder at der må ikke gemmes sessionsdata på serversiden, men derimod på klientsiden. REST sender som standard beskeder i JSON format.

### Token

For at sikre at alle ikke har adgang til API’et, vil det være bedst egnet at have en access token, som kan autorisere brugeren. Access token er et objekt, som indeholder identiteten på brugeren.  Når en bruger logger ind verificerer systemet, brugerens kode, ved at sammenligne det med sikkerheds informationer inde i databasen. Hvis koden verificeres, oprettes et access token.



Figur 4: Applikationens brug af JWT

JSON web Token(JWT) er et JSON Object, der er defineret i RFC7519. Det er en sikker måde og repræsentere et sæt af informationer mellem to parter. Et token består af header, payload og en signatur.

De tre dele for at kunne skabe token:

* **Header**

I headeren definerer man hvilken type token, som man vil gøre brug af, samt bestemmer hvilken form for signatur algoritme man gerne vil benytte.

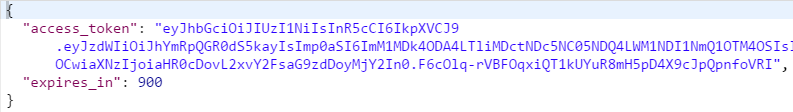
* **Payload**

Består af informationer om den bruger der skal autoriseres. Dette kan f.eks. være brugernavn og kodeord.

* **Signatur**

Signaturen benytter algoritmen der er defineret i headeren til at kryptere header og payload.

For at få et token kobles header, payload og signatur sammen med et punktum som seperator. Token vil på den måde være en kombination der hedder header.payload.signatur, og ser ud som i Figur 5.



Figur 5: Udsnit af token

## Front-end

For at give en bruger mulighed for at kunne se eller arbejde med det data der er blevet hentet og sendt ind i vores database, skal der være en front-end. Denne front-end skal kunne visualisere dataene med et passende grafisk layout, så det er overskueligt at se hvad dataene repræsenterer. Når brugeren har valgt den sensor de vil se data for, skal der være mulighed for at kunne vælge et bestemt tidsinterval, så alt data for en sensor ikke vises på en gang.

### System type

* **Applikation**

Fordelene ved at bygge front-enden som en applikation, er at visualiseringen sker på brugerens computer. Det vil sige hvis brugeren starter applikationen og henter det data der skal bruges, er der mulighed for at kunne se dataene selv om internetforbindelsen mistes. Udover at kunne arbejde offline, vil denne løsning være billig i drift, da der ikke er behov for servere til at have hjemmesiden kørende. Derimod vil der kun være brug for en enkelt server med en database kørende.

* **Hjemmeside**

En anden løsning for front-enden kunne være en hjemmeside. Det smarte ved en hjemmeside er at, den kan tilgås af alle platforme, uden at der skal installeres noget, selv fra en smartphone eller en tablet. Det vil sige at en bruger kun vil skulle åbne sin browser og indtaste en URL til hjemmesiden, og så vil dataene kunne blive vist frem. Da visualisering skal ske på en server, hvor hjemmesiden er installeret, kræver det at der er en server ud over databaseserveren til rådighed. Hjemmesiden skal tilgås af mange brugere på en gang, og der skal have distribueret hjemmesiden ud på flere servere, for at mindske belastningen.

### Programmeringssprog

* **Java**

Fordelene ved at bruge Java som programmeringssprog, er at det fungerer på de fleste platforme, og da der er stillet en Linux server til rådighed, og kunden ønsker at slippe for licenser, vil det være oplagt at bruge Java. Det vil sige at der som udgangspunkt ikke skulle kunne komme nogle problemer under installationen af front-enden på serveren. Ulempen ved at vælge Java som kodesprog, er at det ikke inkorporerer andre sprog så nemt. Det betyder at skal man f.eks. hente noget fra en database, skal man skrive SQL statements. Java har ikke mulighed for at bruge noget der ligner LINQ fra C# .Net.

* **C# .NET**

C# .NET er et programmeringssprog der er designet af Microsoft. Da Microsoft gerne vil have at man bruger alle deres produkter, er de blevet lavet primært til at fungere med hinanden. Det vil sige at hvis man arbejder i C# .NET, fungerer det bedst ved at installere det på en Windows server. Dette kan gøre det mere besværligt at få installeret en C# .NET applikation på f.eks. Linux servere. For at gøre det nemmere er .NET Core blevet opfundet. .NET Core giver mulighed for at lave en ASP.NET hjemmeside eller et ASP.NET API, som kan installeres på f.eks. Linux servere, da det er multi-platform kompatible.  
Fordele ved at vælge C# .NET som programmeringssprog, er at det har LINQ, som giver query-funktionalitet med SQL-lignende syntaks. Dette gør det nemt at arbejde med databaser i C# .NET, da det er som at skrive SQL direkte i C# .NET koden. Ud over dette har alle i gruppen mest erfaring med C# .NET, og gør det derfor oplagt at skulle bruge dette som programmeringssprog.

* **Python**

Python er et programmeringssprog, der er baseret på at skulle være let at forstå, og kunne programmere mere med færre linjer kode. Ud over dette er Python open-source og det fungerer på alle platforme uden specielle programmer. Python fungerer sammen med Bootstrap som er et css bibliotek, der gør det nemt at designe hjemmesiden. Ulemperne ved Python, er at ingen fra gruppen har den store erfaring med dette programmeringssprog, og vi derfor ville være nød til først at skulle lære det.

### Visualisering

* **JQWidgets**

JQWidgets er et JavaScript bibliotek, som man kan tilføje til sit projekt. Det beskriver hvordan grafer skal se ud og hvordan de skal opføre sig, hvis der klikkes eller holdes med musen over grafen. Med JQWidgets er der mulighed for at lave meget mere end grafer, som blandt andet kalendere, knapper, notifikationer, og meget mere. I forhold til grafer, giver JQWidgets mulighed for at oprette 32 forskellige slags grafer, dette vil give os mulighed for at visualisere dataene på en måde så det giver mest mening.  
JQWidgets er samtidig gratis hvis man bruger det i et non-profit projekt. Dette er deres Non-Commercial License. Skal man derimod bruge deres Commercial License koster det fra 199 dollars om året som svare til ca. 1200 kr.[[2]](#footnote-2)

* **Google chart**

Google chart er et gratis JavaScript bibliotek. Det smarte ved google chart er at man ikke skal tilføje biblioteket til ens projekt, men at man refererer til et JavaScript bibliotek på nettet. På denne måde vil biblioteket blive loadet når det skal bruges. Man henter ikke en version af Google chart ned og ligger i sin kode, men man loader derimod scriptsne fra nettet i runtime. Det gør at JavaScript biblioteket altid vil være opdateret, og ved eventuelle fejl, skal applikationen ikke opdateres. Google chart tilbyder mange forskellige diagrammer med et hav af indstillinger. Det er smart da man kan sætte det op så det passer til projektet.

### Sikkerhed

Da det vil være usikkert, at front-end’en sender kodeord til databasen, er det nødvendigt at få dem krypteret inden de bliver sendt videre. Vi har evalueret nogle relevante krypteringsmetoder, for at finde frem til den bedste løsning til vores projekt.

* **3DES**

3DES er en krypterings algoritme der er baseret på DES, den er i dag stemplet som en usikker algoritme. 3DES anses dog for at være sikker, da den har en nøgle størrelse på 112 bits i forhold til DES der har 56 bits. 3DES fungerer på den måde at den løber igennem DES 3 gange ved hjælp af 2 eller 3 nøgler. Hvis der bruges to nøgler krypteres først med en nøgle, dekrypteres med en anden nøgle og til sidst krypteres med første nøgle igen. Dette kaldes EDE (Encryption-Decryption-Encryption). Den anden mulighed hedder EEE, og her bruges 3 nøgler, som skiftes mellem hver kryptering. 3DES gør brug af en symmetrisk nøgle, som vil sige at nøglen skal være kendt for begge parter, da den kan bruges til både at kryptere og dekryptere.

* **AES**

AES er den anbefalede krypterings standard af NIST (National Institute of Standards and Technology), da man kan vælge om den skal have en nøgle længde på 128, 192 eller 256 bit. Valget af nøglelængden har en betydning på hastigheden af krypteringen og dekrypteringen. Som 3DES, er det også nødvendigt for AES, at nøglen er kendt for begge parter. En fordel ved AES er at krypteringen arbejder på hele blokke at tekst, frem for dele af blokke som DES og 3DES gør. Dette giver en større sikkerhed.

* **SHA-2**

SHA-2 er en hashfunktion, der er videreudviklet fra SHA. En hashfunktion giver ikke mulighed for at dekryptere, men derimod kan man sammenligne to krypterede værdier. Det bruges i situationer hvor der kun er en person, der skal kende til værdien, og når den er krypteret kan den kun findes ved at man kryptere den samme værdi og sammenligner dem. Problemet ved hashfunktioner er at man får et mindre output end input. Dette kan gøre at 2 værdier kan ende ud med samme hashværdi. Med SHA-2 kan en besked der skal hashes være 2128 lang før dette kommer til at blive et problem, og da der for det meste kun er tale om kodeord, er det urealistisk at ramme to ens hashværdier.

SHA-2 er erklæret sikker og er anbefalet af NIST.

* **SHA-3**

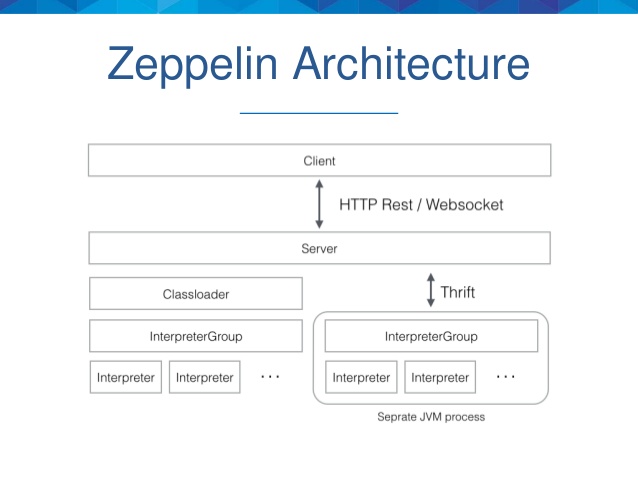
SHA-3 blev opfundet da man mente at SHA-2 snart ville kunne blive brudt. SHA-3 er som SHA-2, en hashfunktion, men SHA-3 er baseret på en ny hash algoritme, kaldt Sponge funktionen. Sponge funktionen giver mulighed for, selv at vælge hvor langt outputtet skal være, og dermed gør at SHA-3 kan håndtere alle størrelser beskeder, uden at de ender med samme hashværdi. Denne hash funktion ses som mere sikker end SHA-2, men ingen af dem er betegnes som usikre.

## Databehandling

I projektbeskrivelsen er der angivet, at det skal være muligt at fremstille data på en pæn visuel måde, samt at kunne arbejde med dataene i form af f.eks. sammenligning. Vi har valgt at analysere følgende værktøjer: Apache Zeppelin, Jupyter og vores egen databehandling, samt visualiserings platform.

### Zeppelin

Apache Zeppelin er en open-source web-baseret notebook platform, som tillader at man kan lave interaktiv databehandling og datavisualisering, samt dele hinandens noter. Zeppelin er Server-Client struktur hvor den kommunikerer med HTTP REST eller Websocket endpoints. En af fordelene ved at benytte Zeppelin er at man kan benytte flere programmeringssprog på samme tid. Det vil være muligt og hente data fra MySQL, hvorefter der kan benyttes R til at lave databehandling og visualisering, på den måde kan gøre brug af den slags kode der får det til at se mest præsentabelt ud. Zeppelin understøtter flere forskellige kodesprog, som blandt andet Python, Scala, Hive, SparkSQL og Shell.



Figur 6: Zeppelin Architecture

### Jupyter

Jupyter er en server-klient webapplikation hvor det er muligt at oprette eller dele dokumenter, som indeholder kode, formler, tekst og grafer. Jupyter notebook tillader at ændre og køre notebooks via web-browseren, men der er også mulighed for, at installere den på en PC uden internet eller på en server, som kan tilgås fra en browser.

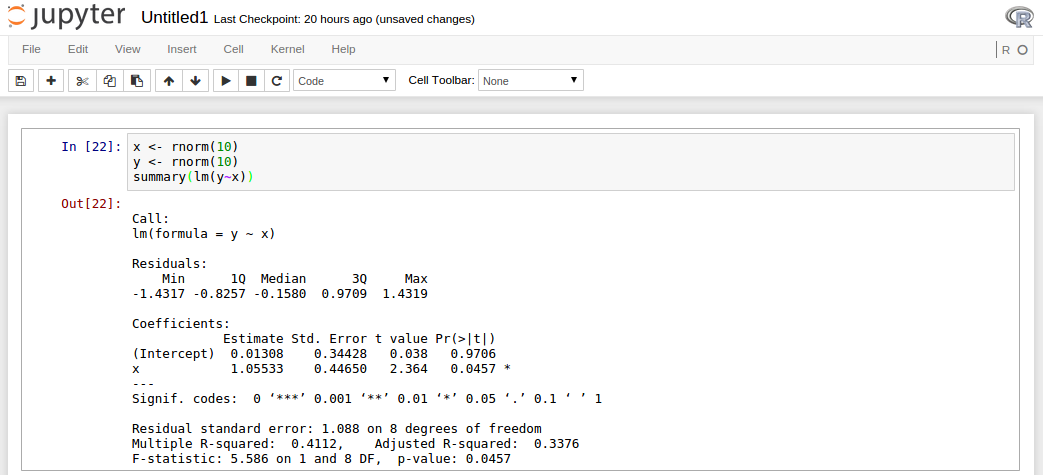
Der findes to primære komponenter i Jupyter:

* **Kerne**

Kernen er et program, der kompiler brugerens kode. Jupyter har en kerne for forskellige programmeringssprog, som bl.a. Python kode.

* **Dashboard**

Dashboard er den visuelle del af applikationen, hvor man kan se sine notebookdokumenter, samt rette i dem. Derudover er der mulighed for at håndtere de kerner, som man gør brug af. Det vil sige at man kan bestemme hvilke som skal køre og hvilke som ikke skal køre.



Figur 7: Jupyter dashboard

### Vores egen platform

Ved at lave vores egen platform vil vi have mulighed for at tage udgangspunkt i hvad, de to andre data platforme kan gøre, og forbedre på de punkter som kan ændres. Platformen skal kunne visualisere vores data og herefter lave databehandling.

Vi vil analysere Apache Spark som et big data værktøj, og hvordan man kan benytte R-kode til at lave databehandling.

Apache Spark er et Open Source Big Data Framework, der designet til at processere data meget hurtigt. Den er baseret på Hadoop Clusteret, men det er ikke en modificeret version af den. Den er blot en måde at implementerer Spark på. Et af hoved fordelene ved anvendelsen af dette værktøj er, at den øger proces hastigheden for en applikation.

Apache Spark indeholder følgende features

* **Spark Streaming**

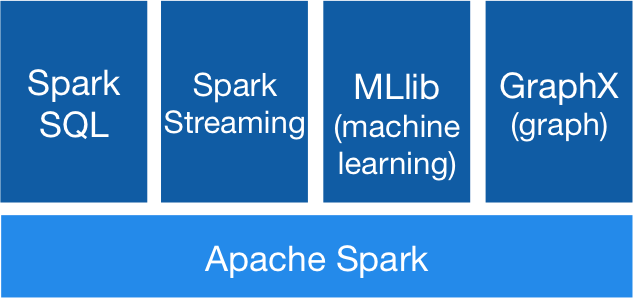
Spark streaming er en udvidelse af API core spank, der gør det nemt og opbygge real time data stream. I platformen kan vi benytte den til at streame data fra sensorerne.

* **Spark SQL**

Spark SQL er Sparks interface, hvor det muligt og arbejde med strukturerede data som JSON datasæt. Desuden kan den skrive og læse data i forskellige formater som JSON.

* **Spark MLib**

Spark MLib er en machine learning som er bygget ovenpå Spark. Den er meget hurtigere en Hadoop diskbaseret.



Figur 8: Illustrere Spark core[[3]](#footnote-3)

Vi kan på hjemmesiden implementer databehandling som kan gøres på to måder. Den ene måde er vha.  Asp.net, hvor vi vil bygge et API som kan håndtere Spark. Desuden kan man yderligere implementerer R programmeringen til applikationen, så det er muligt og skrive koder på hjemmesiden.

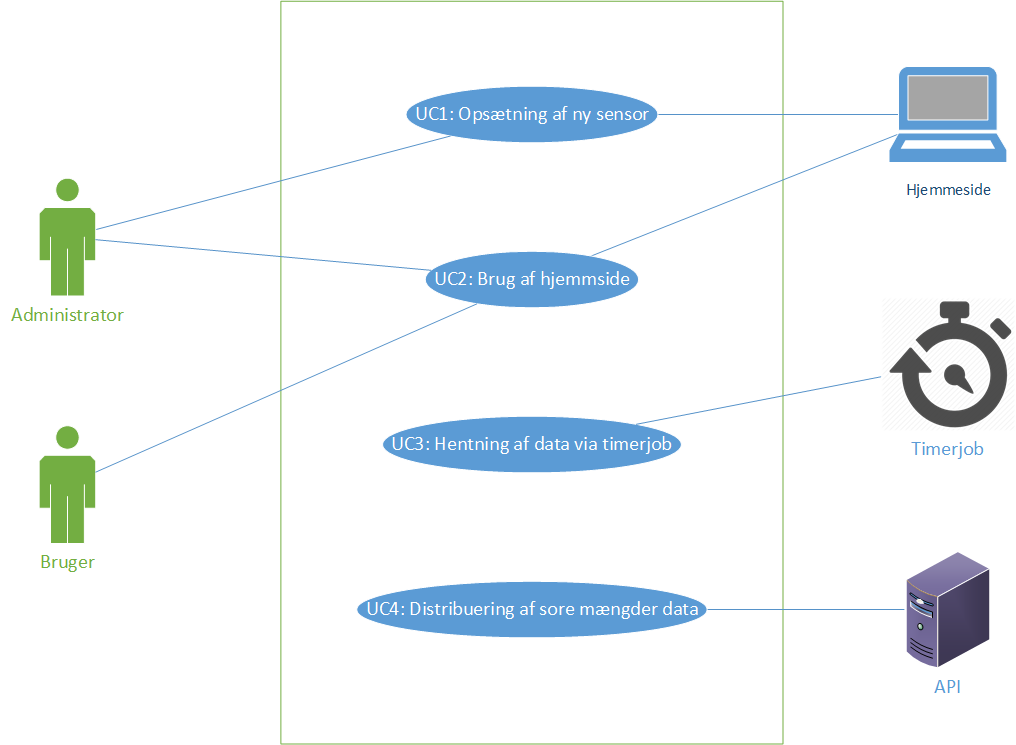
En anden metode kunne være at, man laver et Python API som indeholder metoder til Python Spark.

# Projekt afgrænsning og metoder

## Risikoanalyse

|  |  |  |  |
| --- | --- | --- | --- |
| **Fejl** | **Risiko** | **Påvirkning** | **Løsning** |
| **Databaseserveren melder fejl** | Mellem | Lav | Applikationen skal kunne håndtere fejl fra databasen |
| **Databaseserveren brænder sammen** | Lav | Høj | Der skal dagligt tages backups af databasen |
| **Der bliver sendt data med fejl fra en sensor** | Høj | Lav | Applikationen skal kunne sortere i data med fejl |
| **Applikationsserveren brænder sammen** | Lav | Høj | Der skal være en backup af applikationsserveren |
| **Timerjobbet stopper** | Lav | Lav | Timerjobbet rebootes automatisk |
| **Der sker en fejl på hjemmesiden** | Mellem | Mellem | Hjemmesiden kan håndtere fejl |

## Use case diagram



Figur 9: Usecase diagram

## Brugsscenarier

|  |  |  |
| --- | --- | --- |
| Use case navn | Opsætning af ny sensor | |
| Use case id | **UC1** | |
| Prioritet | Høj | |
| Hovedaktør | Administrator | |
| Beskrivelse | Administratoren skal opsætte en ny sensor | |
| Forudsætning | At sensor er konfigureret korrekt. | |
| Trigger | Administratoren har åbnet hjemmesiden, og har internetadgang. | |
| Normalt event flow | **Aktør: Step 1:** Administratoren trykker på ”Log ind”.  **Step 3:** Administratoren indtaster bruger-id og adgangskode og logger ind.  **Step 5:** Administrator vælger ”Tilføj sensor”.  **Step 7:** Administratoren udfylder de opsætningsvariabler såsom: Sensor type, navn login og værdier som sensor sender. | **System respons  Step 2:** Hjemmesiden viderestiller administratoren til log ind siden.   **Step 4:** Hjemmesiden autorisere brugeren hvorefter der bliver givet administratormuligheder til brugeren.  **Step 6:** Hjemmesiden omdirigere til siden for sensor tilføjelse.    **Step 8:** Sensor bliver oprettet i databasen. |
| Alternativt Forløb | **Flow 1:** Applikationen kan ikke verificere brugeren.  **Flow 2:** Brugeren har ikke administrator adgang. | |
| Konklusion | Administratoren er logget ind, og sensoren er blevet oprettet i databasen. | |

|  |  |  |
| --- | --- | --- |
| Use case navn | Brug af hjemmeside | |
| Use case id | **UC2** | |
| Prioritet | Høj | |
| Hovedaktør | Bruger/Administrator | |
| Beskrivelse | Bruger kan gøre brug af dataene på hjemmesiden | |
| Forudsætning | Bruger er i besiddelse af et produkt det kan gå på nettet | |
| Trigger | Administratoren har åbnet hjemmesiden. | |
| Normalt event flow | **Aktør:** **Step 1:** Brugeren klikker på ’Datakilder’ fanen.  **Step 3:** Brugeren vælger en datakilde og en dato for hvornår dataene skal være fra.   **Step 5:** Brugeren vælger et format, dataene skal eksporteres til og vælger ’Eksporter’. | **System respons:**  **Step 2:** Hjemmesiden viser ’Datakilder’ siden frem.    **Step 4:** Hjemmesiden fremviser data visuelt for brugeren.    **Step 6:** Hjemmesiden henter dataene og downloader det til brugeren enhed. |
| Alternativt Forløb | **Flow 1:** Brugeren har ikke adgang til nogle sensorer.  **Flow 2:** Der er ikke noget data for den angivne dato. | |
| Konklusion | Brugeren har fået visualiseret og hentet data. | |

|  |  |  |
| --- | --- | --- |
| Use case navn | Hentning af data via timerjob | |
| Use case id | **UC3** | |
| Prioritet | Høj | |
| Hovedaktør | Timerjob | |
| Beskrivelse | Nogle sensorer sender data til IC-meter eller ThingSpeak, og systemet skal hente data fra disse. Dette gøres ved et timerjob som periodisk henter data herfra. | |
| Forudsætning | Sensorer sender korrekt data til IC-meter og ThingSpeak. | |
| Trigger | Timerjob skal til at hente data efter et tidsinterval. | |
| Normalt event flow | **Aktør: Step 1:** Timerjobbet finder alle sensorer den skal hente data fra i databasen.  **Step 2:** Timerjobbet forespørger data fra IC-meter / ThingSpeak via deres API.  **Step 3:** Timerjobbet modtager data fra i Json format, og gemmer det i korrekt format i databasen for rå data. Dataene for sensoren ligger nu i databasen og slettes aldrig. | **System respons** |
| Alternativt Forløb | **Flow 1:** Timerjobbet får ikke noget svar, gemmer derfor ingen måling, og skriver en fejl loggen at der ikke blev modtaget noget svar. | |
| Konklusion | Timerjobbet hentede data for hver sensor og gemte målingerne i databasen. | |

|  |  |  |
| --- | --- | --- |
| Use case navn | Distribuering af store mængder data | |
| Use case id | **UC4** | |
| Prioritet | Høj | |
| Hovedaktør | Sensor | |
| Beskrivelse | Formålet er at sikre at serveren ikke bliver overbelastet, samt at det er muligt at API’et, kan håndter data i forskellige formater. | |
| Forudsætning | Databasen er blevet sat op korrekt og at administrator har indsat variablerne for gateway sensoren. | |
| Trigger | At brugeren skal indsætte data via API’et | |
| Normalt event flow | **Aktør:**  **Step 1:** Sensorer sender data i CSV/Json format til API’et i et bestemt interval. | **System respons:**  **Step 2:** Dataene vil nogle gange komme ind i store mængder, og andre gange i mindre, da der kan være mange sensorer. Systemet vil sikre at serveren ikke bliver belastet, vha. af at distribuere forespørgslerne på forskellige server.  S**tep 3:** Systemet vil derefter gemme dataene i databasen. |
| Konklusion | At brugeren har mulighed for at indsætte data i andre formater og gemme i databasen. | |

## Kravsporing

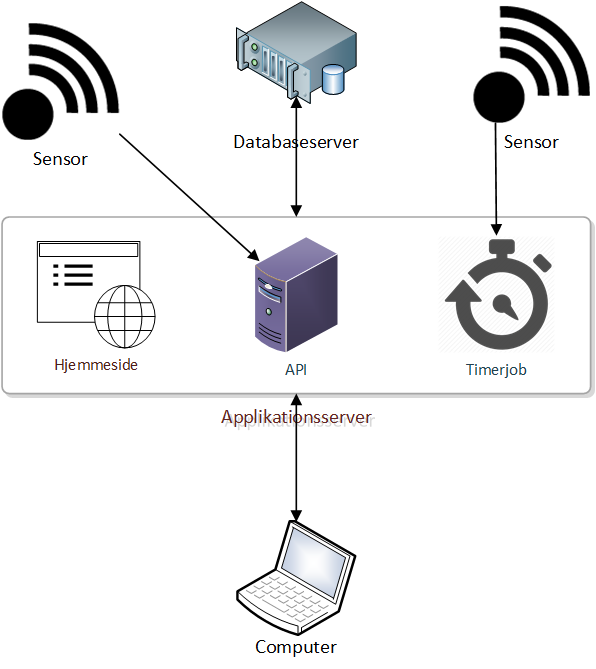
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Krav | UC1 | UC2 | UC3 | UC4 |
| 1: |  |  | X |  |
| 2: |  | X |  |  |
| 3: |  | X |  |  |
| 4: |  | X |  |  |
| 5: |  | X |  |  |
| 6: |  |  | X |  |
| 7: |  |  |  | X |
| 8: |  |  |  | X |
| 9: | X |  |  |  |
| 10: |  |  |  | X |
| 11: |  |  |  |  |
| 12: | X |  | X |  |
| 13: | X |  |  |  |

# Problemløsning

## Server

Vi har valgt at benytte Nginx som web-server og det er baseret ud fra fem vigtige emner, som er performance for statisk og dynamisk indhold, sikkerhed, fleksibilitet, dokumentation og support/forums.  Ud fra disse punkter fandt vi ud af, at den bedste løsning er Nginx fordi, at Nginx har bedre performance.

Apache kan godt håndter forespørgsler, men jo flere anmodninger jo mindre bliver Apaches ydeevne formindsket, men da Nginx er event-baseret har det ingen effekt på Nginx performance og det holder Nginx hukommelsesbrug nede.



Figur 10: Arkitektur diagram

Vi har lavet et distribueret system, hvor vi har to servere. Databaseserveren som ses øverst og applikationsserveren, som indeholder API, timerjobbet og hjemmesiden. En sensor sender dataene ind til API’et, mens timerjobbet henter data fra en anden sensor.

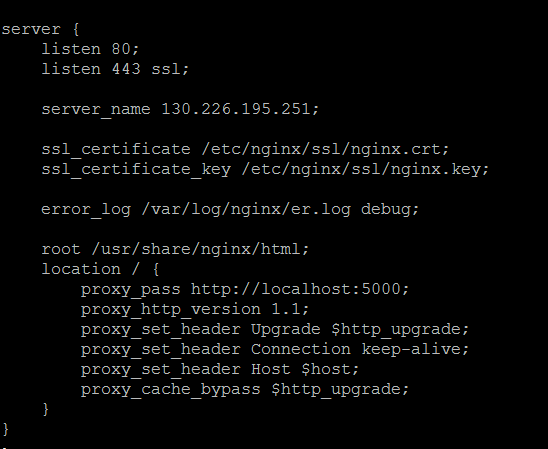
### Nginx opsætning

Serveren der bliver benyttet er en Ubuntu 14.04 LTS, hvorved en packet manager bliver anvendt til at installere Nginx.

sudo apt-get install nginx

Efter installationen kræver det, at der bliver opsat en proxy server til at overføre forespørgsler til ASP applikationen. Ved brug af den metode vil port 80 forwardes til public traffic, og derfra videre til port 5000, således ASP applikationen kan lytte. Herefter installeres der dotnet core, for at kunne køre .NET Core programmer på Linux.

Som det kan ses på Figur 11 lytter vi på port 80.

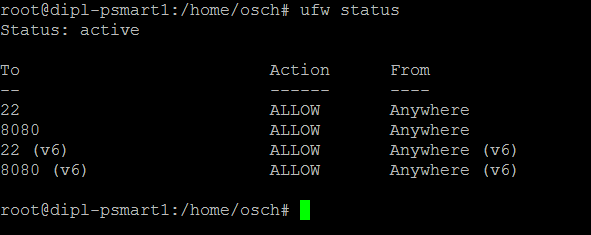


Figur 11: Site-avaliable/default

### Database firewall

I forbindelse med sikkerheden, er serveren delt op i to servere, herunder en databaseserver og en applikationsserver. I forhold til databasen er den opstillet så databasen kun kan tilgås via SSH (Secure Shell).

For at sikre databasens firewall, benyttes der UFW, som er et User Interface til IP-tables.



Figur 12: Firewall

Port 8080 anvendes til MySQL, hvorved der ændres fra default værdien 3306 til 8080.

Dette gøres for at sikre serveren og formindsker risikoen for hacking.

På applikationsserveren er der åbnet for port 8080 til databasens forbindelse samt SSH. Derudover er der åbnet op for port 80(http), for at lave API kald til de forskellige virksomheders API’er. Port 443(HTTPS) er også åbnet, således forbindelsen mellem serveren og applikationen er krypteret.

### SSL certifikat

Da et af kravene er at sikre dataene, har det været nødvendigt at finde en løsning til, hvorledes dataene mellem klienten og serveren kan krypteres. Som standard er forbindelsen mellem serveren og klienten ikke krypteret, hvilket vil sige at folk med adgang til netværket kan overvære alt trafik og inspicere dataene, der er sendt mellem klient og serveren.

MySQL yder krypteret forbindelse mellem klienten og serveren ved hjælp af TLS(Transport Layer Security). TLS er en nyere implementering af SSL.

Der findes to typer af certifikater:

* Self-Signed certificate
* Certificate authority

De besidder begge samme funktioner, som er at kryptere dataene der sørger for en sikker hjemmeside.

Dog er forskellen mellem disse, er at den der hedder Certificate authority fortæller kunderne om alt information på denne server er sikker. Dette kan anses for at være troværdig, da det kommer fra en sikker kilde, hvorimod self-signed certifikat er et certifikat man selv genererer. Udover dette skal projektvejlederen udlevere et DNS, men da der ikke var en DNS tildelt til os, valgte vi at anvende self-signed certifikat. Når der gøres brug af self-signed certifikat bliver det nødvendigt at generere et helt nyt sæt private keys, samt public keys

## Indhentning af data

### Design

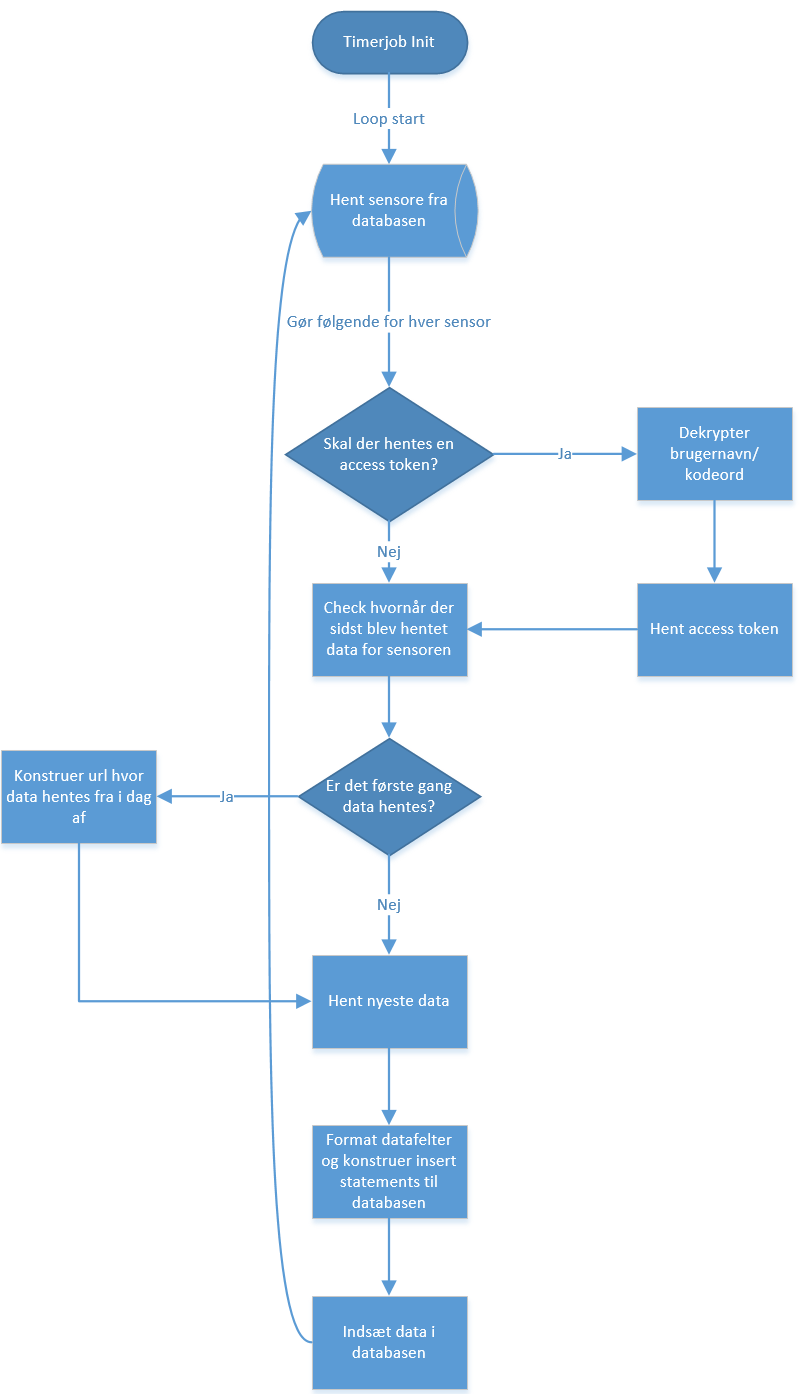
Timerjobbet er lavet i Python, delt op i 3 moduler, som hver især henter data fra hvert sit sted. Den ene henter fra IC-Meter, den anden fra ThingSpeak, og den tredje fra Firebase. Grunden til det er lavet i 3 forskellige moduler er, at dataformaterne er vidt forskellige og måden de hentes på. Derudover skal data ikke hentes med samme tidsinterval på de forskellige sider. Der er også en bedre opdeling af koden, når de ligger i 3 forskellige moduler, og vi kan køre dem som 3 separate processer, som gør det nemmere at fejlsøge, hvis der opstår fejl.

### Implementering

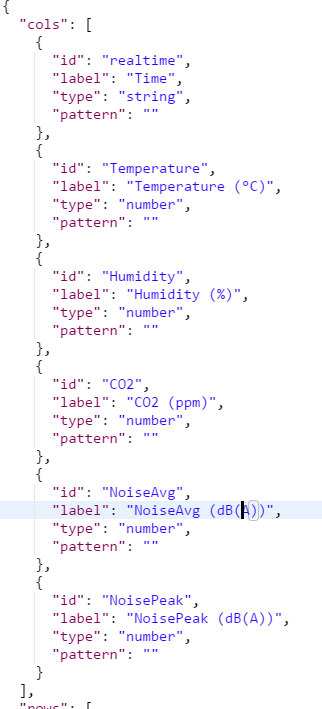
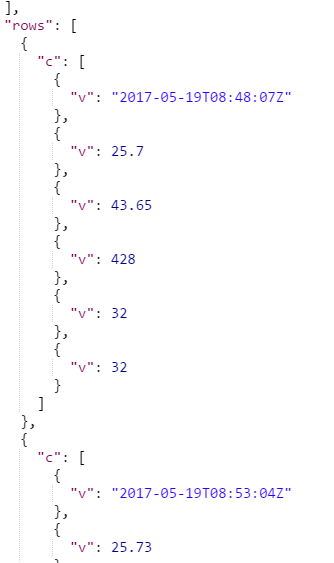
#### IC-Meter

##### Funktionalitet

Data bliver sendt til IC-Meter fra sensorerne hvert 5. minut, og derfor køres et ”loop” af IC-Meter programmet hvert 5. minut. Et IC-Meter loop består af følgende faser:

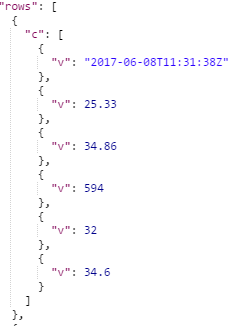


Figur 13: Flowchart for IC-Meter timerjob

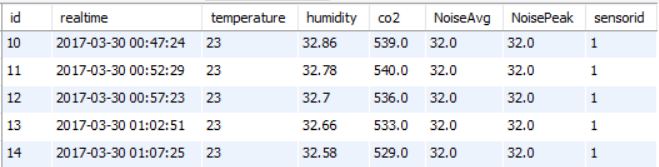
 Det første som IC-Meter modulet gør, er at hente alle IC-Meter sensorer fra databasen. Derefter vil programmet sortere alle sensorerne efter brugere til IC-meter og skabe en sorteret liste af sensorer. Programmet vil herefter køre igennem hver sensor og forsøge at hente data.

Det første programmet gør for hver sensor, er at se om der skal hentes en ny access token. Det skal gøres hvis programmet lige er startet eller hvis sensoren, som er under processering, har en anden bruger end den forrige sensor. IC-Meter delen af timerjobbet, er den eneste del som skal bruge et brugernavn og kodeord for at hente data. Derfor er det også den eneste del der dekrypterer data fra databasen, og det er også grunden til, at vi ikke har hashet kodeordene i databasen. IC-Meter kræver nemlig, at man sender brugernavn og kodeord for at få adgang til sensorerne via en access token.

Skal der hentes en access token, vil programmet først dekryptere brugernavn og kodeord fra databasen via AES kryptering og en hemmelig nøgle. Når den har gjort dette vil den requeste et access token fra IC-Meter via HTTPS. Access token’et gemmes, og programmet næsten klar til at hente data. Vi har et access token og et Box-id for sensoren, og nu skal vi finde ud af i hvilket tidsrum vi vil hente data. Dette finder programmet ud af ved at sende en forespørgsel på hvornår, der sidst blev hentet data for den enkelte sensor i databasen. Er der ingen, vil den bruge dags dato fra kl. 00.00. Er der hentet før, tager den tidspunktet for det nyeste udtræk. Nu vil programmet hente data for sensoren fra IC-Meter via HTTPS. Efter dataene er hentet, vil noget af JSON dataene se ud som på Figur 14:



Figur 14: IC-Meter data i JSON format

Først kommer en beskrivelse af datafelterne, og derefter kommer dataene en efter en i en liste kaldet ”rows”. Det data vi får tilsendt, er alle nye data, da vi har specificeret i hvilket tidsinterval vi vil have data. Data skal nu formatteres rigtigt, og der skal konstrueres en insert statement til databasen. Data felterne kan være forskellige fra sensor til sensor, så der laves dynamisk en insert statement alt efter hvilke felter der sendes. Når data er indsat i databasen kan det se ud som Figur 15.

Figur 15: IC-Meter sensor data tabel

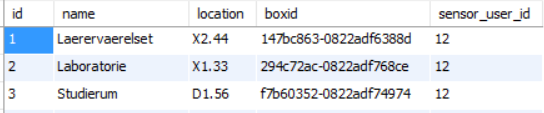
Hvis en sensor ikke sender f.eks. CO2, vil dette felt blive sat til *null*.

Der tages tid på hele processen for at hente data for hver sensor, og programmet starter et nyt ”loop” når der er gået 5 minutter siden det forrige loop startede.

Sker der fejl i et loop for en sensor, vil programmet blot hoppe sensoren over, og dataene vil i stedet hentes i næste loop. Der går på den måde ikke noget data tabt ved at et loop fejler. Er der en fejl i dataformatet, smides det væk, og gemmes ikke i databasen.

Loggen skrives i logfilen, der hedder ”Log.txt”, og eventuelle fejl gemmes i logfilen ”errlog.txt”.

##### Skalérbarhed

Når dataene for en IC-Meter sensor gemmes i databasen, ser det ud som på Figur 16:  


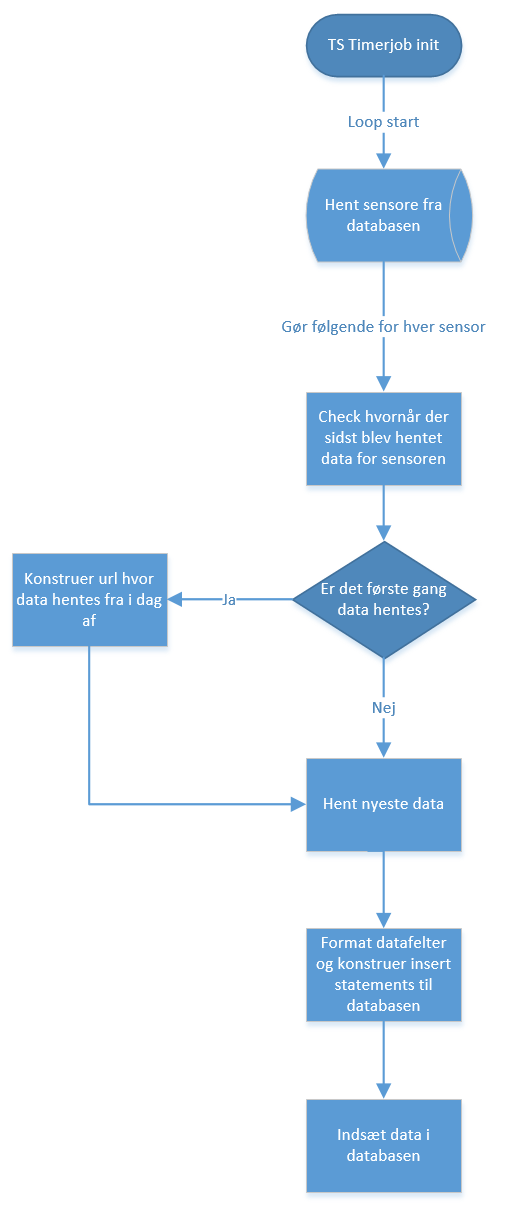
Figur 16: IC-Meter sensor tabel

Vil man tilføje en sensor mere, skal man indtaste et nyt Box-id, en ejer på sensoren, samt et navn og en lokation. Dette gøres alt sammen simpelt på hjemmesiden. Lige nu tager det omkring 3 sekunder pr ”loop”, at hente data for 12 sensorer. Hvis der kommer 2000 sensorer ind, kan man forvente det vil tage omkring 5 minutter at hente alt data, og timerjobbet vil komme bagud i forhold til at hente data. Her skal der enten sættes et højere tidsinterval, eller man skal dele programmet ud i flere tråde.

#### ThingSpeak

##### Funktionalitet

De sensorer der sender til ThingSpeak, sender med forskellige intervaller. Nogle sender flere gange i minuttet og nogle sender et par gange i timen. Vi har derfor valgt at sætte tidsintervallet til hvert 5. minut, da det vil sørge for, at vi har friske data hele tiden.  ThingSpeak modulet skal ikke bruge kodeord og brugernavn, da vi kun kan hente data fra åbne kanaler (public). Vi har derfor ikke nogen dekrypteringsfunktion. ThingSpeak modulets opgaver er illustreret med et flowchart der kan ses i Figur 17.



Figur 17: Flowchart for ThingSpeak timerjob

ThingSpeak timerjobbets opgaver ligner meget IC-Meter timerjobbet. Der hvor der er stor forskel er på dataformatet og fejlhåndteringen. Vi har bemærket, at der oftere sendes fejlmålinger til ThingSpeak, og derfor tjekkes der i højere grad for at det data der er i målingerne, er rigtige værdier og ikke fejlkoder eller tomme felter. Er der fejl eller tomme felter, gemmes den enkelte måling ikke.

ThingSpeak tillader også at man maksimalt sender 8 værdier per måling. Felterne hedder ”field1, field2, field3…” osv. Inde på ThingSpeaks hjemmeside kan man indtaste hvad de forskellige felter er. Derfor gemmes data også efter dette princip. Data der hentes ser ud som Figur 18.   


Figur 18: ThingSpeak data i JSON format

Der gemmes fejl i logfilen ”TS\_errlog.txt” og standard logning i ”TS\_log.txt”.

##### Skalerbarhed

Det eneste der skal tilføjes for at timerjobbet henter data for en ny ThingSpeak sensor er en kanalid og et navn på sensoren. Dette kan gøres via hjemmesiden, og timerjobbet vil med det samme hente data for sensoren. Kanalen skal også være offentlig, for at vi kan få adgang til data.

Lige nu tager det under 1 sekund at hente data for 2 sensorer. Tilføjes der over 2000 sensorer er vores vurdering, at der skal justeres på tidsintervallet eller dele sensorerne op i tråde, ligesom ved IC-Meter timerjobbet.

#### Firebase

##### Funktionalitet

Firebase timerjobbet gør overordnet set det samme som de to andre. Det har dog været noget sværere at udvikle, og processeringstiden er meget højere, da man via vores adgang til firebase kun kan få alt data, som er gemt hos firebase på én gang. Man kan altså ikke specificere det tidsrum, som man gerne vil have data i, og det har gjort det mere besværligt at arbejde med.  Derfor køres det job også kun 1 gang i timen, i modsætning til de andre som kører hvert 5. minut. Når data hentes fra firebase, ser det ud som Figur 19.



Figur 19: Firebase data i JSON format

Hver datablok har et unikt ID, som er autogenereret. Dataene kommer ikke i rækkefølge, og derfor udfører timerjobbet en sortering af alle datablokkene efter tidsstemplet og tjekker op i mode hvornår den sidste måling i databasen er udført. Dette gør den for hver sensor. Derefter vil den sortere alle målinger fra, som er før denne dato, og de resterende vil gemmes. Dataformateringen er også betydeligt sværere, da ID’et på hver datablok er autogenereret, og de ikke kommer i en JSON liste. Vi henter datablokke ved at dele strengen ved hver tuborgklamme, og derefter opstiller det i en liste af nye målinger. De nye målinger kan derefter gemmes i databasen.

Der gemmes fejl i ”FB\_errlog.txt” og standard print i ”FB\_log.txt”.

##### Skalérbarhed

Firebase timerjobbet er mindre skalerbart end de øvrige timerjobs, og dette skyldes at man ikke kan specificere hvilket tidsinterval man vil have data i. Man kan tilføje flere el sensorer ved at indsætte en ny række i el\_sensor tabellen via hjemmesiden.

Her skal man vide hvilken SensorID, som den har på Firebase siden. Dette er ikke noget vi sætter op, men noget som personen, der er ansvarlig for el sensorerne på firebase, kan gøre.

Det tager omkring 10 sekunder at hente og processer data hver time, så der kan tilføjes mange flere sensorer, før timerjobbet ikke kan følge med.

## Opbevaring af data

Vi har valgt en MYSQL database, hovedsageligt fordi vi gerne vil have en struktureret database der giver muligheder for at lave constraints og joins, og fordi vi har gode erfaringer med denne DBMS. Samtidig vil vi spare en masse tid da vi ikke behøver at sætte os ind i hvordan MySQL fungerer.

### ER-Diagram

ER-diagrammet for databasen kan findes i appendiks 11.1, og er designet af vores database. Både IC-Meter, ThingSpeak og Firebase har hver deres entities. Der er en entity som indeholder alle sensorerne for hver af disse sider, og en tilhørende entity som indeholder de målinger vi henter. De sensorer som sender data til os, ”custom sensor”, har også deres egne entities. En til selve sensorerne, en til navnene på de værdier de sender, og en til at indeholde de faktiske værdier de sender.

### RID

RID digrammet kan findes i appendiks 11.2, og viser den færdige database. Hver slags sensor har både en tabel der indeholder alle sensorne og en tabel for målingsværdierne. Den brugerdefinerede sensor ”Custom\_sensor” har 3 tabeller, hvor den tredje er en tabel der beskriver hvilke data den sender.

Der findes både en ”user” tabel og en ”sensoruser” tabel. ”User” tabellen indeholder brugerinformationer for vores egen hjemmeside, og ”Sensoruser” tabellen inderholder bruger informationer for de sider, hvor de kræver, at man logger ind for at hente sensor data.

Brugerne til hjemmeside tilhører grupper, og disse er opdelt i 3 tabeller. En til alle grupperne, en til medlemmer af grupperne, og en som indeholder hvilke sensorer gruppen har adgang til.

## API

Vi har valgt, at bruge et API i projektet, så logikken for systemet er en del for sig selv. Dette gør at systemet bliver bygget op i tre lag (trelagsarkitekturen), databasen, API’et og en front-end. Med trelagsarkitekturen er der mulighed for at skifte et lag ud uden at skulle sortere et andet lag fra. Det vil sige hvis hjemmesiden skulle udskiftes, opbygger man et nyt design, og lader den kalde de samme metoder fra API’et, som den gamle hjemmeside. Derudover kan der nemt implementeres en app, da der kun skal designes en app, og så kan den kalde metoderne fra API’et på samme måde som hjemmesiden.

I projektet har vi valgt at benytte REST og måden vi vil repræsentere dataene vil være i form af JSON.

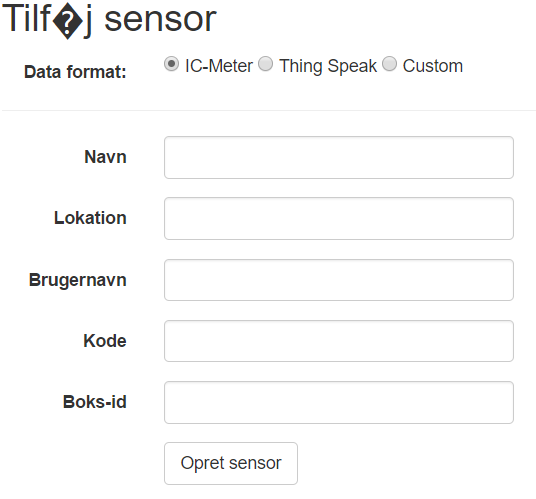
API’et kan udveksle data med databasen så længe der er internetadgang

### Implementering

#### IC-Meter

IC-Meter er en webapplikation som opbevare og visualisere målinger af temperatur, luftfugtighed, CO2 og støj. Vi kan ved benyttelse af API’ets metoder hente forskellige datasæt, som er gemt i databasen.

ICMeterController klassen indeholder metoder til at hente data fra databasen. Der er oprettet en del forskellige metoder, da der både skal hentes, rettes og tilføjes data. Når man logger ind på hjemmesiden og vil tilføje en sensor, vil siden se ud som Figur 20. Her er det muligt at indtaste informationer om en sensor. Denne information vil blive sendt til API’et idet der klikkes på ”Opret sensor” knappen. API’et vil ud fra dette data tilføje en sensor og en sensorbruger til databasen.



Figur 20: Opret IC-Meter sensor fra hjemmesiden

Metoder

|  |  |  |
| --- | --- | --- |
| Request type | Metodenavn | Forklaring |
| GET | ICMeter/getsensors | Henter alle sensorer |
| GET | ICMeter/getSensorsFromGroupIds | Henter bestemte sensorer ud fra et gruppe id |
| GET | ICMeter/getSensor | Henter en bestemt sensor ud fra et id |
| POST | ICMeter /addSensor | Tilføjer en ny sensor |
| GET | ICMeter/getValues | Henter sensor målinger ud fra et id |
| GET | ICMeter/getValuesFromDates | Henter sensor målinger ud fra et id og en start og slut dato |
| POST | ICMeter/addSensorToGroup | Giver en gruppe mulighed for at se sensoren på hjemmesiden |
| GET | ICMeter /AllData | Få alle data fra ICmeter |
| GET | ICMeter/sensor/{sensorName} | Få en specifik sensor type(Luftfugtighed) |

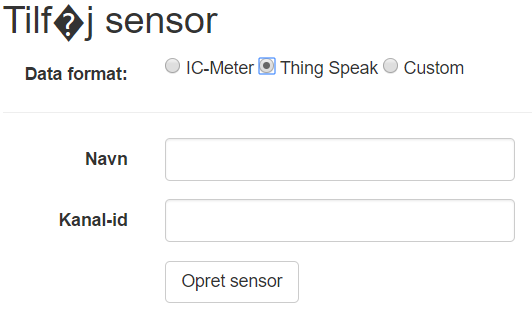
Figur 21: IC-Meter API metoder

#### ThingSpeak

ThingSpeak er en IoT platform (Internet of Things), der gør det muligt at indsamle, opbevare, analysere, visualisere og arbejde med data fra sensorerne. I projektet benytter vi de sensorer som kunden har tilføjet til ThingSpeak, som primært er parkeringspladsen og indendørsklima.

ThingSpeakController klassen gør det samme som ICMeterController klassen, med den undtagelse af at den henter, tilføjer og ændrer i tabellen ts\_sensor og ts\_values frem for ic\_sensor og ic\_values.

Et eksempel på hvordan det ser ud når man vil tilføje en ThingSpeak sensor på hjemmesiden kan ses på Figur 22.



Figur 22: Opret ThingSpeak sensor fra hjemmesiden

Metoder

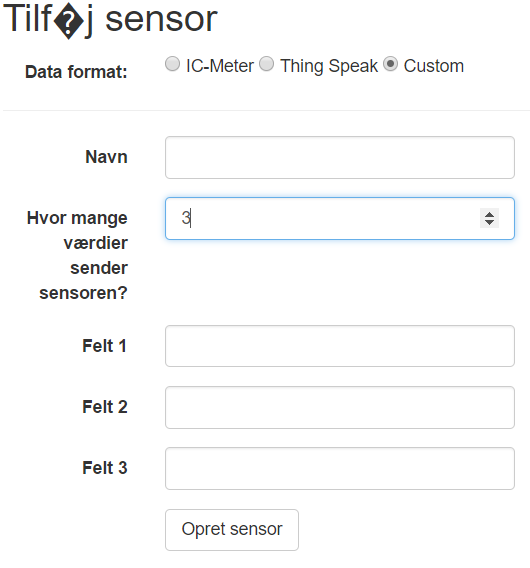
|  |  |  |
| --- | --- | --- |
| Request type | Metodenavn | Forklaring |
| POST | ThingSpeak/addSensor | Tilføjer en ny sensor |
| GET | ThingSpeak/getSensors | Henter alle sensorer |
| GET | ThingSpeak/getSensorsFromGroupId | Henter bestemte sensorer ud fra et gruppe id |
| GET | ThingSpeak/getSensor | Henter en bestemt sensor ud fra et id |
| GET | ThingSpeak/getValues | Henter sensor målinger ud fra et id |
| GET | ThingSpeak/getValuesFromDates | Henter sensor målinger ud fra et id og en start og slut dato |
| POST | ThingSpeak/addSensorToGroup | Giver en gruppe mulighed for at se sensoren på hjemmesiden |

Figur 23: ThingSpeak API metoder

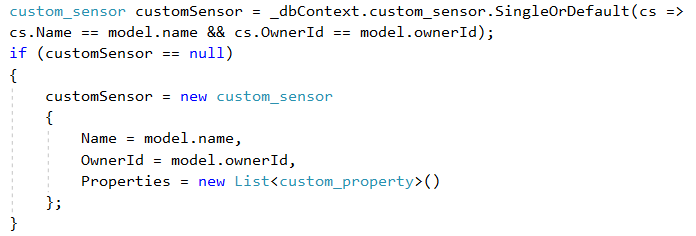
#### Gateway

Formålet med gatewayen er, at man skal kunne tilføje sensorer direkte til vores system, frem for at dataene skal gå igennem et andet firma som IC-Meter eller ThingSpeak før vi får det. Gatewayen implementeres generisk. Det vil sige at der ikke er et bestemt antal datafelter der skal udfyldes. Hvis en sensor kun måler temperatur, eller hvis den måler 10 forskellige ting, så vil gatewayen kunne håndtere det.

Når en sensor skal tilføjes til vores system, skal den først tilføjes på hjemmesiden. Et udklip af hjemmesiden kan ses i Figur 24. På hjemmesiden kan man definere hvad navnet på sensoren skal være og hvor mange datafelter sensoren sender. Alt efter hvilket antal datafelter man vælger, vil et tilsvarende antal tekstbokse vises. Når sensoren er oprettet gennem hjemmesiden, skal den sættes op til at sende data til API’et. API’et kan ud fra et id kæde dataene sammen med sensoren i databasen.



Figur 24: Opret custom sensor fra hjemmesiden



Figur 25: LINQ’s benyttelse for at gemme sensoren i databasen.

##### Uploading af data

Måden hvorpå vi har valgt at oprette nye sensorer i databasen er ved at benytte en custom\_sensor-, custom\_property- og custom\_sensor-tabel. Navnet på sensoren bliver gemt i custom\_sensor, sammen med navnet på den der har oprettet sensoren. Custom\_property vil gemme de datafelter som sensoren sender (fx C02 eller temperatur). Den sidste tabel benyttes til at gemme de værdier sensoren vil registrere med et tidsstempel. Alle tabellerne bruger ID’er til at relatere til hinanden.



Figur 26: Illustrere hvad tabellerne indeholder

Når sensoren er sat op og sender til API’et, bruger den API metoden*CustomSensor/addJson*. Alt dette sker i formatet JSON. Da det ikke er API’et der kalder sensoren, men omvendt bliver det muligt for brugeren af sensoren, selv at vælge hvor ofte data skal sendes. Lige efter API metoden er kaldt, er der mulighed for at gå ind på hjemmesiden og se en graf med det samme.

Metoder

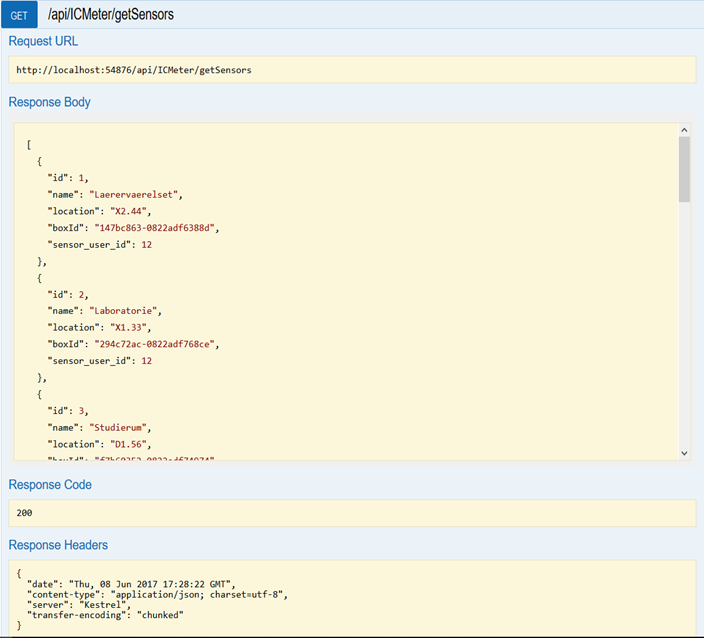
|  |  |  |
| --- | --- | --- |
| Request type | Metodenavn | Forklaring |
| GET | CustomSensor/getsensors | Henter alle sensorer |
| GET | CustomSensor/getSensorsFromGroupId | Henter bestemte sensorer ud fra et gruppe id |
| GET | CustomSensor/getSensor | Henter en bestemt sensor ud fra et id |
| POST | CustomSensor/addSensor | Tilføjer en ny sensor |
| GET | CustomSensor/getProperties | Henter datafelter ud fra en sensors id |
| GET | CustomSensor/getValuesFromDates | Henter sensor målinger ud fra et id og en start og slut dato |
| POST | CustomSensor/addSensorToGroup | Giver en gruppe mulighed for at se sensoren på hjemmesiden |
| POST | CustomSensor/addJson | Tilføjer data til databasen ud fra et JSON format |

Figur 27: Gateway API metoder

### Swagger

Swagger er et open source værktøj der benyttes til API dokumentation. Swagger beskriver strukturen af vores API, så det er læsbart. Swagger er implementeret i API’et, så det er muligt at køre swagger på hjemmesiden, så vi kan se dokumentationen

I Figur 28 er et eksempel på hvordan en metode i Swagger ser ud.



Figur 28: Metode vist i swagger

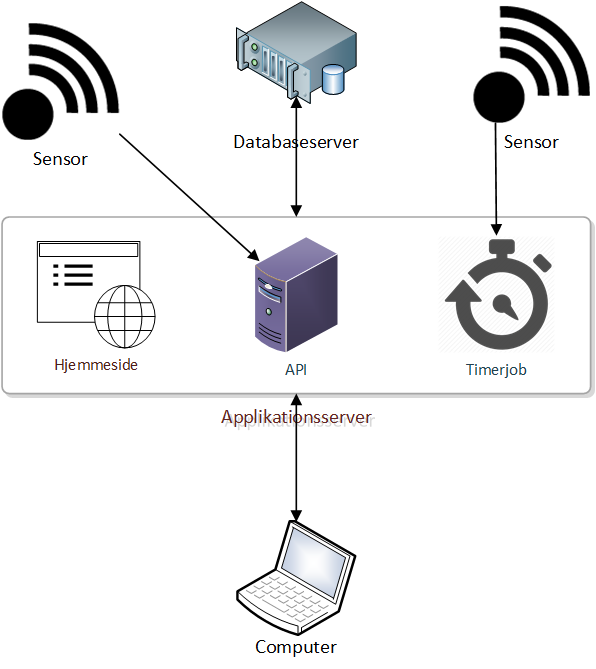
### Token

Da token ikke var ønsket af kunden, er det kun blevet implementeret, så man kan kalde API’et og få et token, men der bruges ikke token til hverken at hente eller sende data til og fra databasen.

## Front-end

### Systemtype

Da det var et krav at løsningen skulle være mulig at tilgå fra alle platforme, var det oplagt at lave en hjemmeside frem for en applikation, der installeres på brugerens computer. Med en hjemmeside vil det være muligt at gå ind i browseren fra hvilken som helst enhed og skrive URL’en ind for at tilgå løsningen. Dette gælder både for en Linux, Windows eller IOS maskine. Der vil endda være mulighed for at tilgå den fra en smartphone. Ulemperne ved denne løsning er, at der skal stilles flere servere op, da der komme mange brugere på samme server på samme tid. Dette kan ende med at en server vil blive overbelastet og til sidst ende med at lukke ned. For at undgå et nedbrud, skal belastningen fordeles ud på flere servere, som på den måde bliver dyrere at vedligeholde.



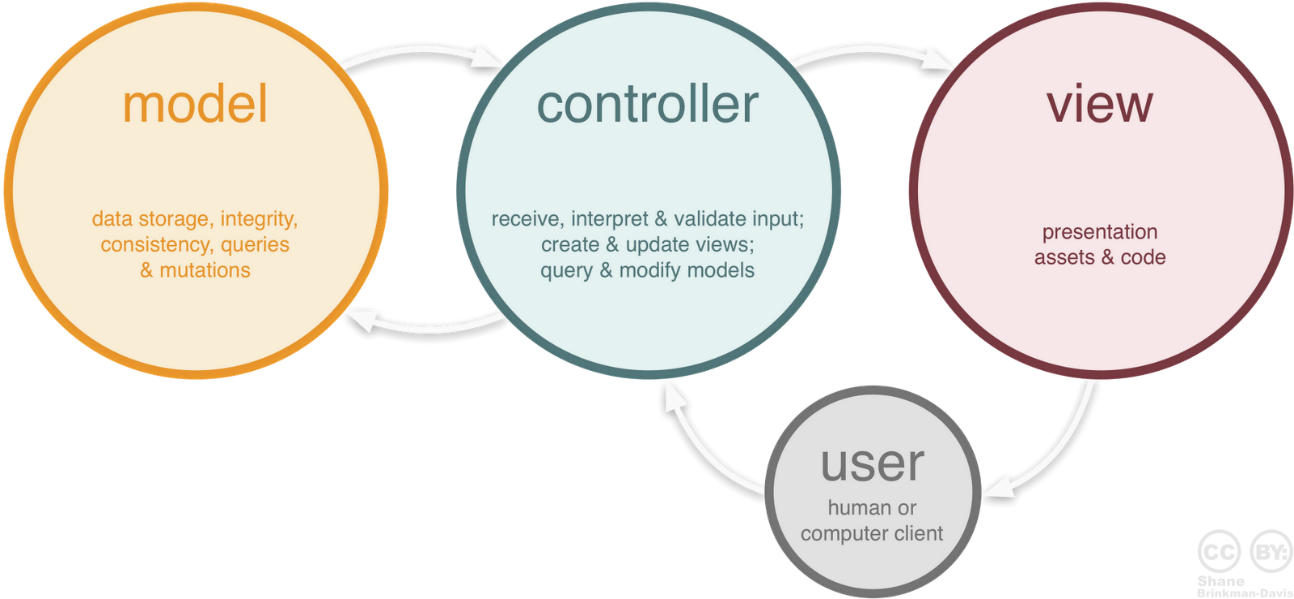
Figur 29: Arkitektur diagram

Som der ses på Figur 29, er hjemmesiden en del af applikationsserverne. Der vil være mange applikationsservere, som alle indeholder hjemmesiden og API’et. Dette er med til at gøre driften dyre end, hvis brugeren skulle installere applikationen på sin lokale maskine. Dette vil gøre det muligt at fjerne applikationsserverne, så der kun skulle opsættes en databaseserver.

### Programmeringssprog

Det har været svært at komme frem til hvilket programmeringssprog der skulle bruges, da et af kravene er, at løsningen skal køre på en Linux server. Programmeringssproget er dog faldet på C# .NET, da det er hvad gruppen har mest erfaring med. Hvis man ser på Java, er det et par år siden nogle fra gruppen har arbejdet med det og Python har ingen arbejdet med før. Den eneste måde hvorpå C# .NET kan fungere på en Linux maskine, er ved at bruge C# .NET Core, som ikke har meget til forskel fra almindelig C# .NET. C# .NET Core er meget nyt (Versionsnummeret er 1.1 i forhold til almindelig C# .NET der er 4.7) og der er derfor ikke mange funktioner og 3. parts biblioteker, der fungere med C# .NET Core. Dette har gjort det svært at vælge hvilke 3. parts biblioteker vi skulle bruge.

Da C# .NET kun er programmeringssproget, har vi valgt at bruge ASP.NET som framework til vores hjemmeside, da det er C# .NET’s web framework. Strukturen på hjemmesiden er bygget op i MVC, som betyder Model-View-Controller. Dette er en struktur, der gør koden mere overskuelig, da man har delt koden op i tre dele. Den ene af delene er modellen, der indeholder klasser, som repræsentere objekter. Vi har i vores projekt fx en klasse kaldt LoginModel, der indeholder variablerne fx e-mail og kode. Viewet er den grafiske del, delt op i mindst en klasse for hver side, som kan ses i browseren. I nogle tilfælde vil viewet udfylde en model, som vil blive sendt med i controlleren. Controlleren er den sidste del, som er den der står for logikken. Det vil sige hver gang der klikkes på noget på hjemmesiden, vil controlleren tage stilling til hændelsen og efterfølgende fortælle viewet hvad der skal gøres. Strukturen for MVC er visualiseret i Figur 30



Figur 30: MVC struktur

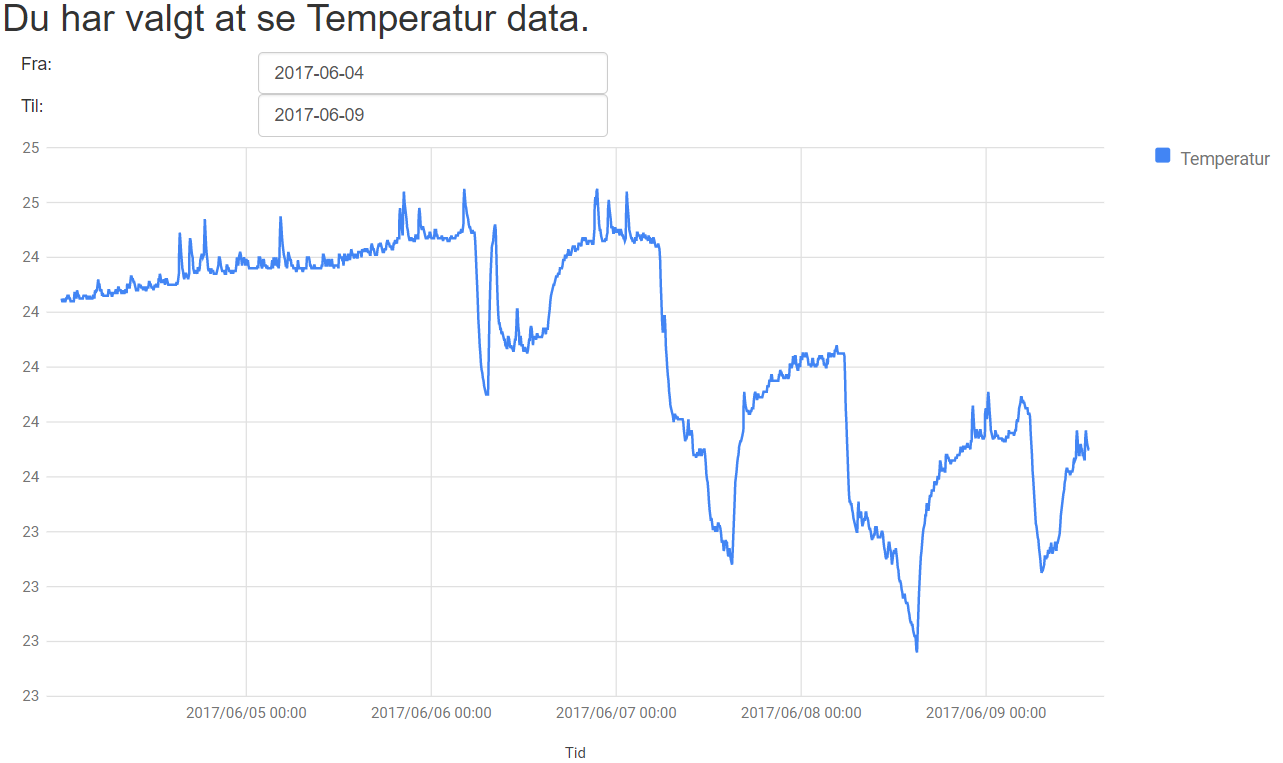
### Visualisering

Først havde vi valgt at bruge JQWidgets til at vise dataene frem, men vi løb ind i mange problemer, som fx data med tidsstempler mellem 12 og 13 ville blive placeret mellem 00 og 01, mens alle andre ville blive placeret rigtigt. Vi har op til flere gange mailet til JQWidgets selv, for at få hjælp, men de var ikke til stor hjælp, da de ikke kunne genskabe fejlen hos dem selv. Det betyder at vi endte ud i at prøve Google chart i stedet for. Google chart var nemmere at sætte op og lettere at komme i gang med. Samtidig var der mange flere der havde arbejdet med det og havde stillet de samme spørgsmål som os. Det vil sige at vi let kunne google os frem til svar, hvis vi skulle komme ind i problemer, eller hvis der var ting vi ikke vidste hvordan skulle løses.

Google chart bliver brugt, når brugeren har valgt en datakilde og hvilket datafelt fra datakilden der skal vises. Første gang brugeren vælger datakilde og datafelt, vises data fra dags dato, men over grafen er der en kalendervisning der gør det muligt at vælge en til og fra dato for det data man vil se. Google chart tilpasser sig efter det data den får ind, som vil sige at hvis der er en dag, hvor der kun er data fra kl. 00:00 til kl. 15:00, så viser den ikke et tomt diagram fra 15:00 til 24:00, men derimod kun frem til kl. 15:00. Samtidig tilpasses højden automatisk, så det højeste punkt i diagrammet er det højeste punkt i dataene, og det sammen med det laveste punkt.

Google chart gør det muligt for brugeren, at holde på grafen og få den specifikke værdi fra der hvor man holder musen over. Dette er en smart funktion, da man ikke altid kan måle helt præcis på en graf når der er mellem 200 og 300 punkter på x-aksen.

Der kan ses et eksempel på en graf på Figur 31.



Figur 31: Graf over CO2

### Sikkerhed

Hjemmesiden er endt med at have flere forskellige slags sikkerhed. Første gang man klikker ind på hjemmesiden, kan man oppe i højre hjørne se en log ind knap, der giver brugeren mulighed for at logge ind på hjemmesiden. Har man ikke nogen bruger, kan man oprette sig som ny bruger og her er der en hashfunktion der hasher kodeordet inden det bliver sendt videre fra hjemmesiden.

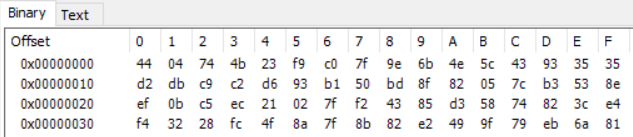
Valget af hashfunktion er faldet på SHA-2 (nærmere SHA-512) da SHA-3 ikke er blevet implementeret i C# .NET Core endnu. Hvis SHA-3 havde været implementeret i C# .NET Core havde vi valgt at bruge denne i stedet, da NIST mener at denne er længere fra at blive brudt end SHA-2. Når koden er blevet hashet, bliver den sendt ned i databasen, hvor den er ulæselig for alle. Det eneste denne værdi nu kan bruges til er når man vil logge ind igen. Her skriver man sin e-mail og kode ind, og koden vil igen blive hashet, hvorefter den sammenlignes med den der findes i database. Hvis de to ikke matcher er det fordi det er den forkerte kode, og brugeren vil ikke blive logget ind. Ligesom ved oprettelse af en bruger, bliver koden hashet inden den bliver sendt væk fra hjemmesiden.

Kigger man i databasen vil det se ud som man kan se på Figur 32.



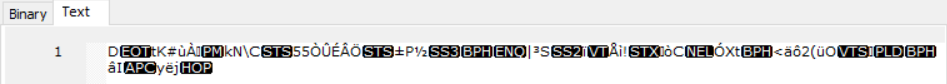
Figur 32: Database udklip af en bruger

Her kan det ses at der står ”BLOB” i password kolonnen. Dette betyder at det er en binær værdi. Vælger man at åbne den binære værdi, vil det se ud som på Figur 33.



Figur 33: Binær værdi af password

Vælger man at se værdien som almindelig tekst, vil det se ud som på Figur 34, og dette giver intet hint om hvad koden skulle være.



Figur 34: Tekst værdi af password

Når man er logget ind på hjemmesiden, bliver der tjekket hvilke grupper man er medlem af, så der vides hvilke sensorer der skal vises for brugeren. Dette giver mulighed for administratorer at oprette en sensor, som de ikke vil have, at alle brugere skal kunne se. Samtidig bliver der tjekket om brugeren er en administrator. Som administrator har man adgang til at tilføje sensorer og ændre på gruppe indstillinger, i form af at oprette nye grupper, og tilføje sensorer og brugere til de grupper administratoren er ejer af.

Hvis admin vælger at tilføje en sensor, får han/hun mulighed for at vælge mellem IC-meter, ThingSpeak eller custom sensor. Hvor hvis der bliver valgt IC-Meter, skal man indtaste brugernavnet og kodeordet til login på IC-Meter. Da dette er følsomt data er der implementeret en symmetrisk nøglealgoritme til at kryptere både brugernavn og kodeord inden det bliver sendt videre fra hjemmesiden. Valget af symmetrisk nøglealgoritme er faldet på AES, da denne har en større nøgle størrelse end 3DES

## databehandling

#### Zeppelin

Grundet mangel på ressourcer til udvikling af vores egen platform valgte vi i gruppen at gøre brug af andre alternativer. Zeppelins formål er, at det skal kunne anvendes som en front-end til applikationen.

Her skulle der gennem Zeppelin kunne hentes og visualiseres data, men da Zeppelin som standard kun tilbyder notebooks, hvor man skal have lidt erfaring med en at de indbyggede kodesprog, ville det ikke være så brugervenligt. Derfor var idéen at vi ville videreudvikle Zeppelin, sådan at der var en del der kunne benyttes ved hjælp af knapper. På den måde ville en bruger der ikke havde noget erfaring med kodning kunne logge ind på siden, finde sin sensor og se data for den, samtidig med at en bruger der har forstand på kodning ville kunne lave databehandling på dataene i databasen.

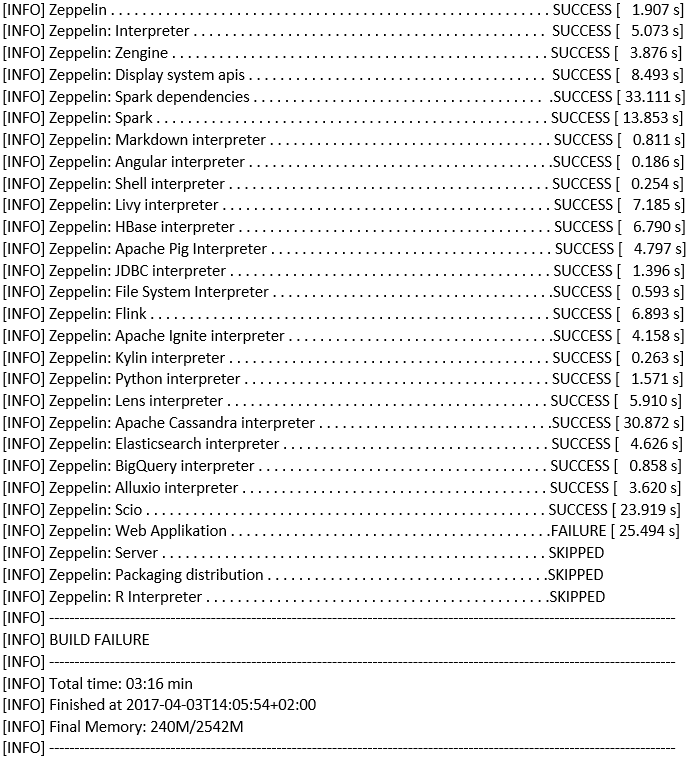
##### Zeppelin installationen

I forbindelse med installationen havde vi oplevet problemer med installationen af Zeppelin idet vi har haft kæmpet med at få sat det op på serveren. Zeppelin installationen har to forskellige slags installationsfiler, den ene er en binær fil, som bar installere Zeppelin, og den anden er source fil, som gør det muligt at videreudvikle på Zeppelin, ved at give adgang til alle filer. Vi valgte at prøve og installere Source filen, da det er open-source og derfor giver mulighed for at ændre i koden, så vi kunne implementere den brugervenlige del.

For at kunne installere source filen har det krævet, at der var installeret følgende programmer på vores Ubuntu server.

|  |  |
| --- | --- |
| **Navn** | **Værdi** |
| **Git** | (Any Version) |
| **Maven** | 3.1.x or higher |
| **JDK** | 1.7 |
| **OS** | 16.04 |

Figur 35: Hardware minimumskrav

Efter at have hentet source filen via Git, skulle man installere det ved brug af Maven, som er et bygge management værktøj for .Java filer, som bliver kompileret til .class filer. Herefter skal Maven have kompileret filerne og der var her at fejlen opstod for os. Som det kan ses i Figur 36 fejler installationen ved installation af webapplikationen. Vi fandt ud af at Zeppelin ikke kunne kører på Ubuntu 16.04 endnu. Der kan ses i Figur 37 , hvilke miljøer Zeppelin kræver. Vi prøvede efterfølgende at få fat i en Ubuntu 14.04 server, hvor vi satte det op på samme måde som med Ubuntu 16.04 og efter at have gjort det, virkede det stadig ikke. På Ubuntu 14.04 fik vi dog ikke helt samme fejlkode. Nu var grunden til at det ikke virkede, at vi ikke havde installeret Node eller Yarn. Men selv efter at have installeret både Node og Yarn, sagde den at den ikke kunne lokalisere installationerne.    


Figur 36: Zeppelin installationsfejl

|  |  |
| --- | --- |
| **Name** | **Value** |
| **Oracle JDK** | 1.7  (set JAVA\_HOME) |
| **OS** | Mac OSX  Ubuntu 14.X  CentOS 6.X  Windows 7 Pro SP1 |

Figur 37: Miljø minimumskrav

Da vi ikke kunne få source filen til at fungere, valgte vi at prøve den binære fil i stedet for. Med denne installeret på serveren, ville vi kunne lave vores egen hjemmeside med et link til Zeppelin, der har forbindelse til vores database. *På denne måde vil der stadig være lavet en brugervenlig del og en del hvor der er mange muligheder for databehandling.*

For at installere den binære fil, skal man hente installationsfilen fra Zeppelin download side. [[4]](#footnote-4) Man vælger et sted på serveren, til at pakke filen ud. Derefter installeres der Spark på serveren og porten til Zeppelin skal åbnes, det kræver at man gøre det vha. UFW og derefter man skriver ”zeppelin-daemon.sh start” i kommandolinjen. Derefter er Zeppelin klar til brug.

# Test

## Server

Forløbet for at starte front-enden og API’et, sker gennem terminalen. Vi har valgt at gøre det med acceptance test, da det er den mest optimale måde at test denne del.

For at få adgang til serverne kræves der SSH forbindelse, hvor der skal logges ind på serveren med en public key.

Step 1:

Åben terminalen og naviger til mappen som indeholder front-end eller API projektet.

https://lh6.googleusercontent.com/UWRcvcEREBFK20PumDiGtoVyR2H61KNfGan9AmISGK26PuFxt3UvuQSJY5-_eS_mxQo9Mv5MjFWJ6xcd1nfKApt7TQNPIga9P86Gl9aD3388zyNM1rJXW7wZ8gCQSWPDA8Zb0UsJ

Figur 38: Destination for API

https://lh5.googleusercontent.com/9b3DxYq11kanddQ2S0WnwMtkSVNf49588cQF5Dhpzh__GQWik_7UJ28FSN_rMClTxmq6o8mvD9mUmTqL_DrLPbpu5q4R3xeH3PaSRlP_cRYJH7WlOmAJMvVE0PN9NFjCXHUnZneG

Figur 39: Destination for front-end

Step 2:

Første gang projektet skal køres, skal det først kompileres ved at skrive kommandoerne ”dotnet restore” og ”dotnet build”.

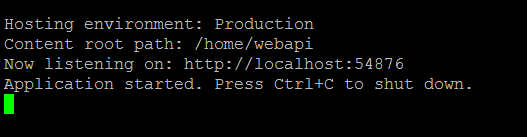
Efter følgende skrives kommandoen ”nohub dotnet run &” der gør at front-end eller API kører konstant.

https://lh6.googleusercontent.com/8GzwRqhga8VE5i5z9y9qrppSACBjpa539_vH_Rav2iC5nUEBnGu4dFcPq9bNCAf197mUNEqg3LW_74bTa2t8tyq08bVic6EmIEGc0L6UgH32cG80EXC-uhWN4L8_-99vCurMQA5K

Figur 40: Kommando for at køre et dotnet program

Step 4:

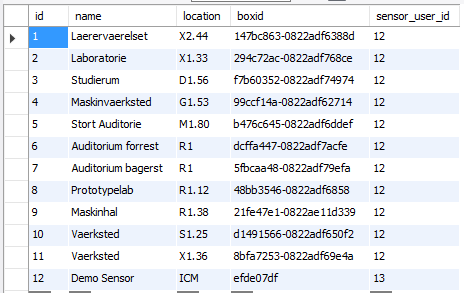
Systemet kører både API og front-enden.



Figur 41: Udklip af log fra dotnet program

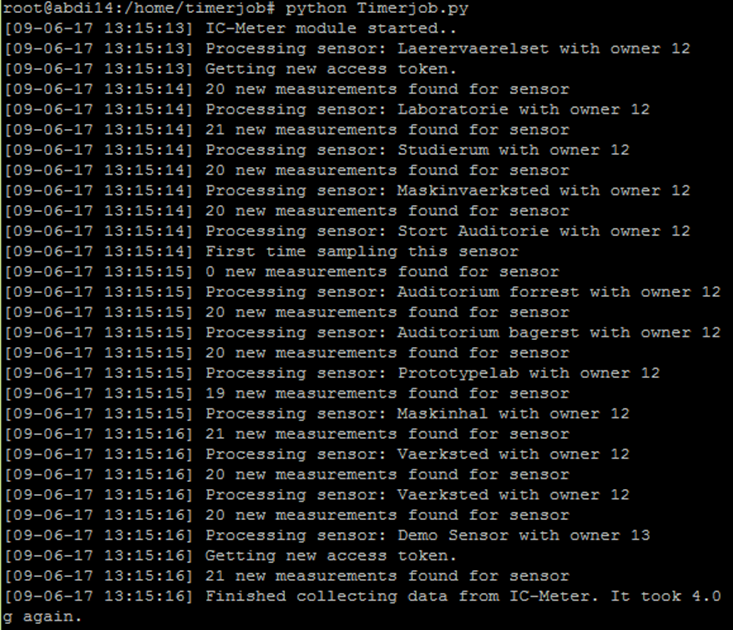
## TimerJob

Acceptance test use case 3  
  
Step 1: Timerjobbet finder alle sensorer den skal hente data fra i databasen.  
I øjeblikket indeholder databasen følgende 12 sensorer for IC-Meter:



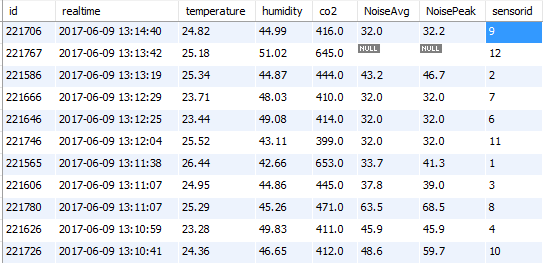
Figur 42: IC-Meter sensor tabellen

Timerjobbet skal altså hente data for 12 sensorer.   
Timerjobbet startes op:



Figur 43: IC-Meter sensor processering

Som det kan ses i Figur 43 blev der processeret 12 sensorer, og timerjobbet har derfor korrekt hentet alle sensorer.   
  
  
Step 2: Timerjobbet forespørger data fra IC-meter / ThingSpeak via deres API.  
Step 3: Timerjobbet modtager data fra i Json format, og gemmer det i korrekt format i databasen for rå data. Dataene for sensoren ligger nu i databasen og slettes aldrig.  
  
I ovenstående billede blev der hentet omkring 20 målinger ved hver sensor, og timerjobbet har derfor korrekt forespurgt data fra IC-meter og gemt det i databasen.   
I nedenstående billede kan vi se i databasen at der er hentet nyt data for alle sensorne, på nær den ene som der ikke var noget nyt data for. Vi har derfor vist at timerjobbet henter alle sensorerne korrekt fra databasen, og indsætter data for dem alle i databasen.

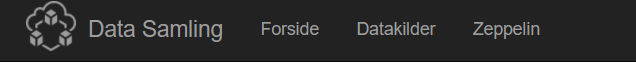


Figur 44: IC-Meter values tabellen

Databehandling testning

Acceptance test af zeppelin

Step 1 Brugeren klikker på zeppelin I hovedmenuen



Figur 45: Udklip af topbaren fra hjemmesiden

Step 2

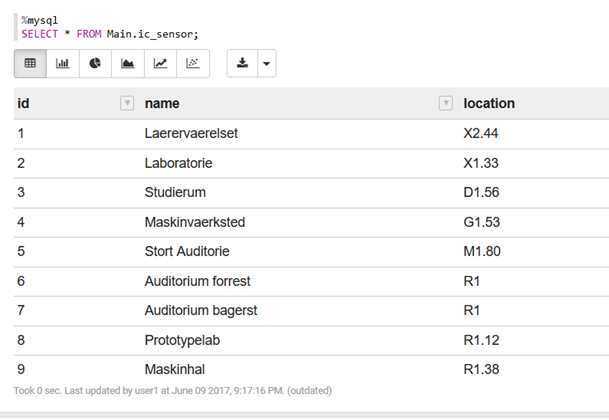
Brugeren logger ind med username og password



Figur 46: Udklip af login siden på Zeppelin

Step 3

Hjemmesiden kan fremvise dataene



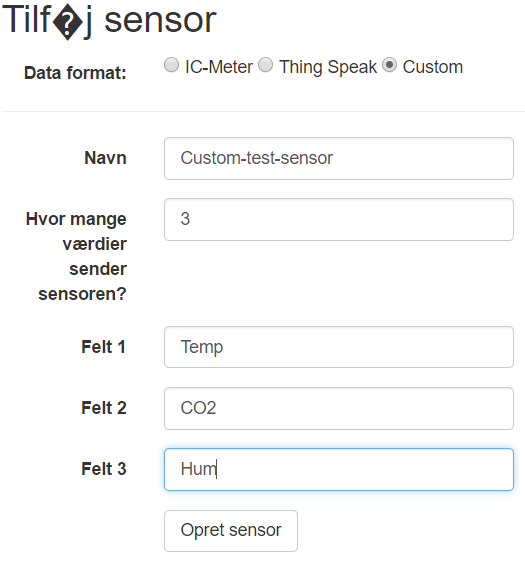
Figur 47: Udklip af Zeppelin notebook

## API

Testning af den generiske Gateway er opbygget således at vi har valgt at illustrere, hvordan sensoren oprette. Det givne data uploades ved hjælp af API’et, hvorved det efterfølgende visualiseres. Gennem dette fokuseres der på beskrivelsen af, hvorledes der trinvis oprettes en sensor via hjemmeside og API.

Der er udarbejdet en console-applikation til at demonstrere gateway.

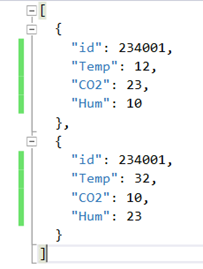
1.   Til dette oprettes en sensor via hjemmesiden



Figur 48: Tilføj custom sensor - udfyldt

Figuren ovenover illustrerer hvordan vi opretter en sensor.

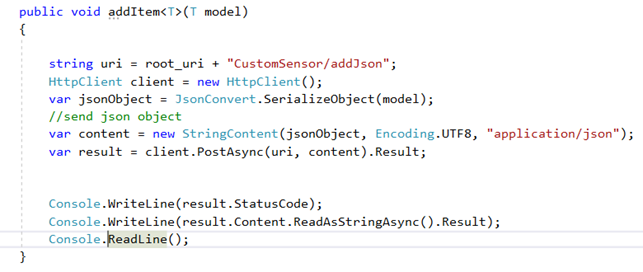
2.   Herefter benyttes console-applikationen til at udføre et api kald til Customsensor/AddJson metode som indeholder vores dummydata.



Figur 49: Custom JSON data

Console applikationen indeholder dummydata, som medfører at api kaldet gemmer dataene i databasen løbende.

Udførslen af api-kaldet kan ses på nedenstående figur.

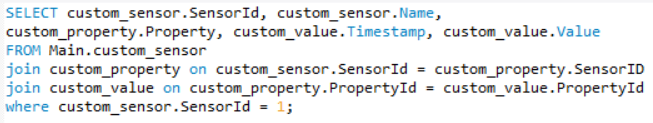


Figur 50: Udsnit af koden, der tilføjer JSON til databasen

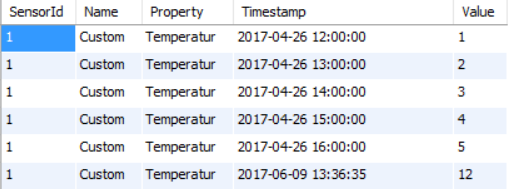
3.   Api-kaldet er konfigureret til at køre hvert andet minut.

Figuren nedenfor viser oprettelsen af sensor med det indgående data samt tidspunkt.

Database SQL

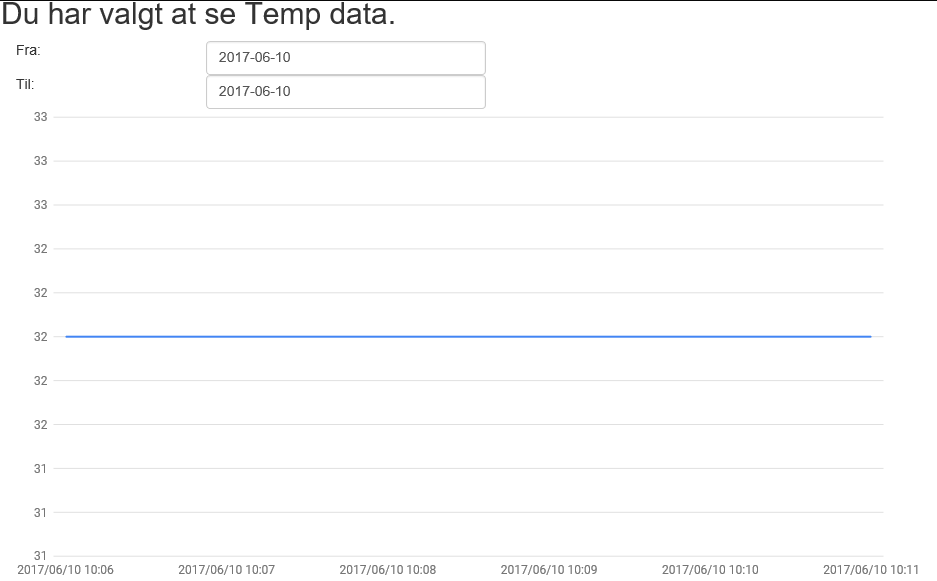


Figur 51: SQL-query til custom sensor



Figur 52: Datasæt for SQL-query

4.   Dernæst illustreres der med hjemmesiden sensorens graf.



Figur 53: Graf for custom sensor

Acceptance test af distribuering af data med flere sensorer.

I denne acceptance testes der når der flere sensorer der benytter det samme API kald

|  |
| --- |
| **Step 1** Sensorerne  sender data til A  **https://lh6.googleusercontent.com/DRLeTl-CdgXLhmJkcXdzujTyqc96tJ_vEjbLVI1ITKS4s1fiA-6i2T6vZUrxUU0KKa7Qe3dky6z26eAOPjhztOUxf10Sso2SuS0OeDqyvnc_301FFMmjXR6oe1V6QGmAqv2-qmJx**  **Step 2:** Dataene vil nogle gange komme ind i store mængder, og andre gange i mindre, da der kan være mange sensorer. Systemet vil sikre at serveren ikke bliver belastet, vha. af at distribuere forespørgslerne på forskellige server.    https://lh5.googleusercontent.com/3iUdhQ21260FyKQxUIZ2djEhaTkSe6XKVKUx58R_FGtC0qyf0NAqq4uJb0xdZGccVYwuO31CEbrj_i64KyV7fLw-HcNFSVImpkLdtPuO4DfRbUXNEBjPBCOYWrF6T78yGPS6sNvW  S**tep 3:** Systemet vil derefter gemme dataene i databasen.    https://lh5.googleusercontent.com/vjRZ3Y1DZd2eN2cVm8SjaLEIJ7WXv_ypjeFTmjfcWg2yY9sR0pXjTNZChfVjisFMgfbWwI0EzHLevYOS89cjBLRz9sLsScUYD_7dXsfFAktuo67xF1jsy95V37fIV8E9e9vjBLPh |

## Front-end

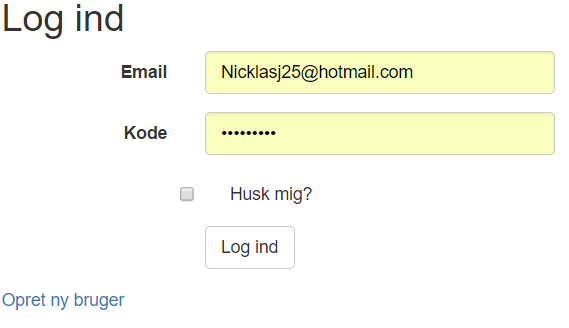
### Acceptance test

Front-enden er lige så vigtig at have testet, som resten af systemet. Dette er den, da en front-end ikke vil fungere uden database eller API, men API eller database vil være ubrugelige uden en front-end.

Front-enden er testet, med acceptance test, hvor der er valgt at tage dele af gangen. Det vil sige at hver side er blevet sat op til at gøre hvad der ønskes den skulle gøre, hvorefter der er blevet indsat forskellige værdier, klikket på alle knapper og alt bliver prøvet af. På denne måde kan man teste om hjemmesiden fungere som den skal.

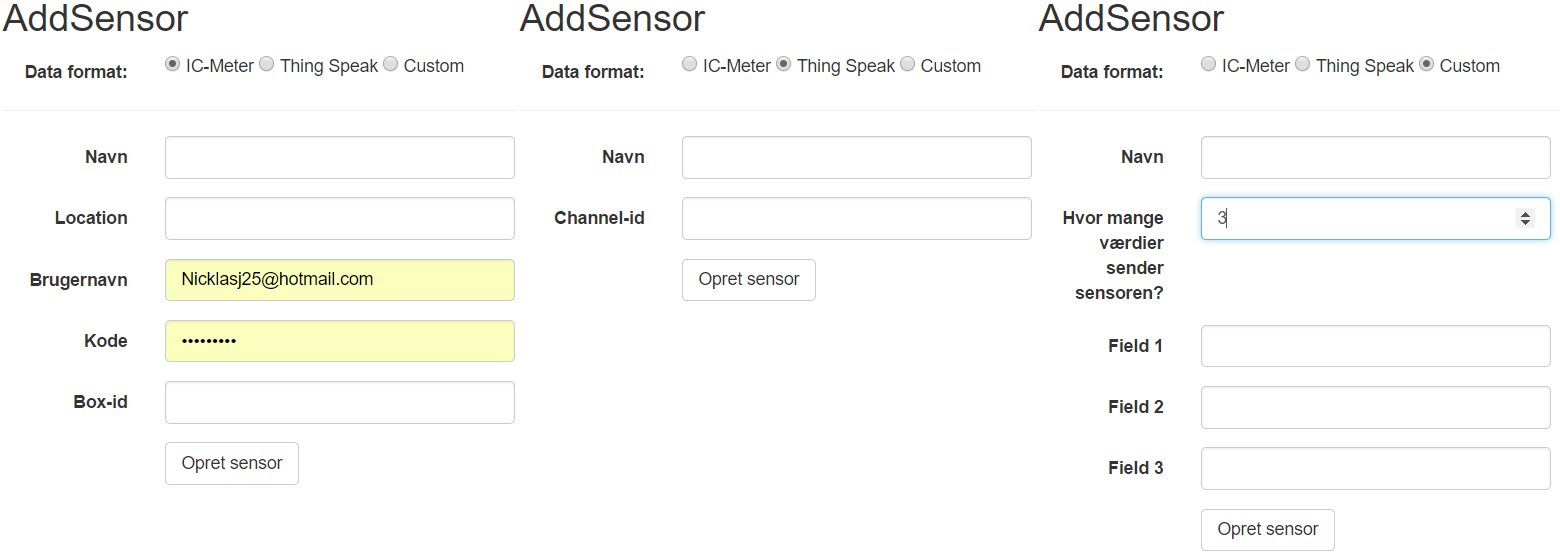
I de fleste tilfælde følger man en eller flere usecases, da usecases er det man ønsker systemet skal kunne. Hvis man kan komme igennem en usecase uden fejl fungere systemet som ønsket. Eksempelvis var første usecase at en administrator kan logge ind på hjemmesiden og tilføje en ny sensor. Her bliver login funktionen og sensor tilføjelse testet.

Første step i usecasen er at brugeren logger ind. Dette gør brugeren på loginsiden, som kan ses i Figur 54.



Figur 54: Loginsiden

Når brugeren er logget ind som en administrator, får han/hun mulighed for at vælge “tilføj sensor” i menuen oppe i højre hjørne. På “tilføj sensor” siden får brugeren en masse valgmuligheder i form af, hvilken type sensor skal tilføjes, hvad er navnet på sensoren, hvad er datafelterne m.m. Eksempler på dette kan ses i Figur 55.



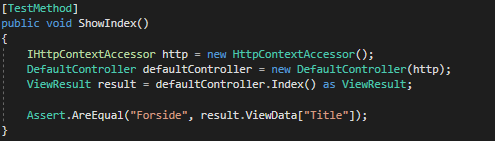
Figur 55: Tilføj sensor siden

Når brugeren har indtastet de nødvendige informationer, bliver sensoren oprettet i databasen. Det kan efterfølgende konkluderes at use casen fungerer, hvis sensoren kan findes under datakilder siden.

Da acceptance test er at klikke rundt og bruge hjemmesiden på almindelige og ualmindelige måder, kan man gøre brug af det der hedder Money test. Monkey test er når man får et program til at klikke tilfældigt rundt på hjemmesiden. En måde at bruge Monkey test på er ved at klikke ind på <https://monkeytest.it>, her indtaster du linket på din hjemmeside og 5-10 minutter efter svare den tilbage om den har fundet nogle fejl.

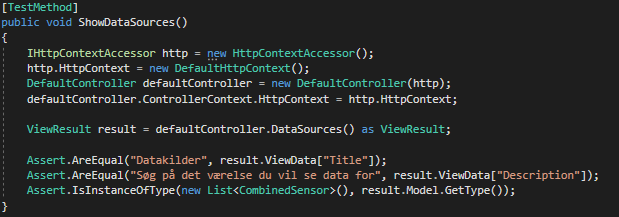
### Unit test

Da man ikke kan være helt sikker, på at komme gennem alle scenarier når der bruges acceptance test, har vi valgt også at lave unit test på front-enden. Dette gøres ved at man tager controlleren og sender forskelligt data gennem alle metoderne. På denne måde får man testet mest muligt igennem.



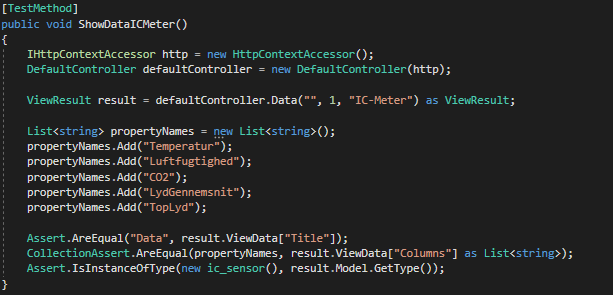
Figur 56: Test af Index()

I Figur 56 testes Index metoden fra controlleren DefaultController. Denne metodes opgave er at sende brugeren til forsiden af hjemmesiden, samtidig med at den definere titlen på siden. I testen startes der med at initialisere controlleren sådan at metoderne kan kaldes. Når man kalder metoden får man svaret tilbage som et ViewResult. Dette giver mulighed for at se modellen der bliver sendt retur, samt ViewBag, som er valgfrie ting man kan sende fra controlleren til view’et. Til sidst kommer Assert, som er den del der vurdere om metoden er forløbet som forventet. Det første argument er hvad man forventer, og det andet argument er hvad der kommer retur fra metoden.



Figur 57: Test af DataSouces()

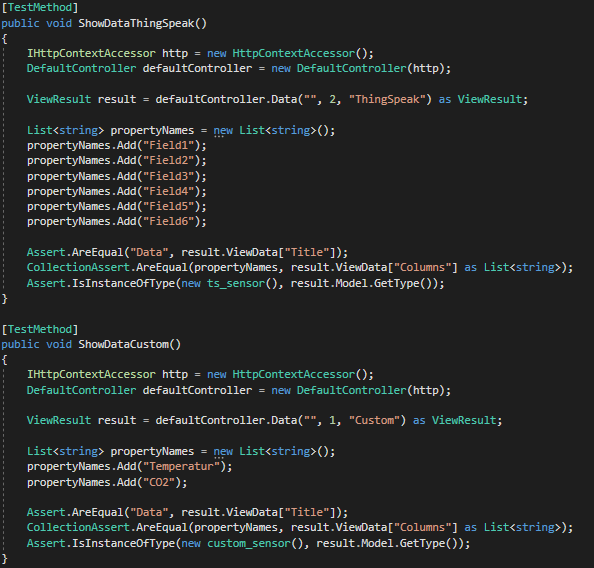
I Figur 57 testes metoden DataSources, som lige som metoden Index, sender brugeren til en ny side. Samtidig finder den ud af hvilken bruger der er logget ind og ud fra dette, henter den en liste over de sensorer der skal vises for brugeren. Som det kan ses i Figur 57 er der mere der skal testes end i metoden Index. Først testes der om ViewBag, er som forventet. Efterfølgende testes det om modellen er af samme type som List<CombinedSensor>. Disse typer sammenlignes ved at kalde Assert.IsInstanceOfType.



Figur 58: Test af Data med IC-Meter som input

En tredje metode der hedder Data, modtager inputsne data, sensorid og sensortype. Data, er den slags data der er valgt for den specifikke sensor (f.eks. Temperatur eller CO2), sensorid er id’et for sensoren og sensortype er om sensoren er en IC-Meter-, ThingSpeak- eller custom-sensor. Alt efter hvad værdi der er i sensortypen, kommer der forskellige forløb.

I Figur 58 er Data metoden testet med sensortype sat til ”IC-Meter”. Dette gør at metoden henter en IC-Meter sensor, og de datafelter der hører til. Da IC-Meter datafelterne altid er temperatur, luftfugtighed, CO2, Lydgennemsnit og toplyd, vælges der at oprette en liste med disse, som der kan sammenlignes med, den liste som metoden Data returnere. Da Assert kun sammenligner to enkelte værdier og der skal sammenlignes to lister af værdier, bruges der CollectionAssert. CollectionAssert tager to lister og sammenligner alle værdier fra den ene liste med alle værdier fra den anden liste.



Figur 59: Test af Data() med ThingSpeak og Custom som input

Da Data metoden kan have to andre scenarier, hvor inputtet sensortype er ThingSpeak eller Custom, skal dette også testes igennem. I Figur 59 kan man se at den eneste forskel fra sensortype IC-Meter, er at listen Columns indeholder andre værdier, og modellen er a typen ts\_sensor eller custom\_sensor i stedet for ic\_sensor.

Ved at teste controlleren på denne måde, kommer man igennem alle metoder, på alle de måder man har lyst til. Unit test giver også mulighed for at se om værdier returnere null, om de er false eller true, og mange andre ting, som gør at alle outputs kan blive vurderet.

# Konklusion

## Produktvurdering

Det kan konkluderes at produktet lever op til største delen af vores krav. I følgende vil vi vurdere kravene enkeltvist.

1. **Alt data skal ligge i databasen, uden at det på noget tidspunkt vil blive slettet.**

Databasen er endt med at være delt op, sådan at det kan håndtere data fra både ThingSpeak, IC-Meter, direkte fra sensorer m.m. Databasen er sat op sådan at data fylder så lidt som muligt, og gør derfor at data, let kan blive opbevaret uden at skulle slettes på noget tidspunkt.

1. **Webløsningen skal kunne tilgås fra alle platforme.**

Da front-enden er designet som en hjemmeside er der mulighed for at kunne tilgå den fra alle platforme der har en browser. Hjemmesiden er ikke designet til at kunne håndtere små skærme, som vil sige at mobilversioner ikke giver mulighed for at vælge et punkt i topbaren. Dette kan løses ved at åbne siden som computerversion når den åbnes på en mobil.

1. **Webløsningen skal illustrere dataene på en spændene måde.**

Ved hjælp af Google Chat, har vi fået illustreret dataene i en flot graf. Grafen har mange indstillinger, som gør at grafen kan rettes mere til, alt efter hvordan man helt præcis vil have den. Samtidig har vi Zeppelin kørende hvor der er mulighed for selv at hente data og få det illustreret.

1. **Webløsningen skal muliggøre at brugere kan sammenligne data**

Zeppelin er blevet installeret på serveren, hvor der kan findes et link til i toppen af hjemmesiden. Da Zeppelin giver mulighed for at arbejde med data, kan man sammenligne en til flere grafer.

1. **Det skal være muligt at kunne hente dataene i flere formater fra webløsningen.**

Hvis man klikker ind på hjemmesiden og finder den graf man vil hente data fra, giver hjemmesiden ikke denne mulighed. Det er dog muligt at åbne Zeppelin og gennem noget kode hente det data man vil have i hvilket som helst format.

1. **Timerjob skal kunne hente dataene periodisk, fra ThingSpeak og IC-Meter.**

Udover ThingSpeak og IC-Meter er timejobbet endt ud i at kunne hente data fra Firebase. De er sat op i forskellige intervaller for hentning af data. ThingSpeak henter data hver 5. minut. IC-Meter henter lige som ThingSpeak hvert 5. minut. Firebase henter en gang i timen.

1. **Systemet skal kunne håndtere variabel indkomne datamængder.**

Da systemet både kan håndtere at der hentes data og at der bliver sent data til systemet, har vi fået timerjobbet til at kunne klare at der bliver tilføjet flere sensorer. På den måde vil der komme mere og mere data der skal hentes. Gatewayen er den der modtager data, og kan også håndtere at der komme data fra flere sensorer. Der er en grænse for hvor meget de kan håndtere, men denne grænse ligger højt.

1. **Systemet skal kunne håndtere Excel-data**

Da vi ikke har været i besiddelse af Excel-data, eller har følt at det var noget der var relevant, har vi valgt at nedprioritere dette krav. Det betyder at systemet ikke kan håndtere data i form af et Excel-ark.

1. **Systemet skal indeholde en administrations side, hvor der kan håndteres autorisering.**

Hjemmesiden er end ud i at man skal være logget ind for at kunne tilgå en administrations side. Administrations siden giver mulighed for administratoren at oprette nye grupper, hvor der skal vælges om gruppen er en administrator gruppe. Når sensorer er oprettet i databasen kan man tilføje disse sensorer til forskellige grupper, sådan at det kun er bestemte personer der kan se den. Når der kommer nye brugere til systemet kan administratoren tilføje disse brugere til de grupper de skal have rettigheder.

1. **Systemet skal kunne håndtere variation af belastning.**

Som systemet fungerer nu, kan det ikke håndtere for meget belastning, da der kun er en applikationsserver. Systemet er dog designet sådan at der er mulighed for at stille flere applikationsservere op, og ved hjælp af en load balancer kan distribuere belastningen. Dette vil gøre at systemet kan håndtere flere brugere.

1. **Der skal analyseres hvorvidt systemet kan skaleres.**

Systemet er skalerbart i den forståelse af at databasen kan udvides med ekstra plads, når den på et tidspunkt bliver udfyldt med data fra sensorerne. Som sagt i tidligere punkt, kan man skalere systemet ved at tilføje flere applikationsservere og implementere en load balancer der kan distribuere belastningen.

1. **Det skal være muligt at registrere nye sensorer til system via en gateway.**

På hjemmesiden er der mulighed for en administrator at gå ind på en side, hvor de kan tilføje nye sensorer. Ud over IC-Meter og ThingSpeak sensorer, kan man tilføje en custom sensor, som er systemets gateway. Når custom sensoren er tilføjet på hjemmesiden, kan man sætte sin sensor op til at sende data til API’et, som vil ligge det i databasen. Når dataene ligger i databasen, giver det mulighed for hjemmesiden at vise dataene frem i grafer, og at lave databehandling gennem Zeppelin.

### Sikkerhed

#### Server

Vi har vurderet at sikkerheden på serveren er fint, da vi har sikret os at serverne separeres fra hinanden. For at kunne have sikret serveren endnu mere, skulle vi have specificeret ip-adressen fra applikationsserveren, således risikoen for at serveren bliver hacket reduceres. For at kunne have sikret os imod ud fra kommende kunne vi have implementeret Intrusion Detection System(IDS), på den måde kunne vi have sikret os imod, hvis der have været ændret ved filerne. Hvis filerne var ændret kunne vi så have haft modtaget en mail, som så vil have notificeret administratoren, om at serveren er blevet ændret. Vi har yderligere også sikret databasen, da vi har tilføjet SSL til databasen samt har vi også sikret hjemmesiden, da vi har valgt at benytte HTTPS med vores egen certifikat.

#### Database

Den største sikkerhed vi kunne give databasen ville være at dekryptere alt der ligger i databasen. På denne måde kan en ubuden gæst ikke se hvad der er i databasen. Dette ville dog også gøre systemet langsommere, da der skal dekrypteres hver gang der skal hentes data, og da vi har med big data at gøre, er dette heller ikke optimalt, da vi henter og sender data til databasen hele tiden. Det er ikke ønsket af alle at kryptere alle data, men det kunne dog være en god ide at give brugerne mulighed for at vælge at gemme datasen krypteret hvis de ønsker det. Dette kunne være hvis der bliver gemt fortrolige oplysninger.

#### Timerjob

Mens vi har udviklet timerjobbet har vi tænkt på sikkerhed, men da er stadig nogle områder af timerjobbet, som kan forbedres sikkerhedsmæssigt. Vi har krypteret kodeord og brugernavn i databasen, som timerjobbet benytter, så de ikke ligger i plaintekst. Disse kodeord og brugernavn dekryptere vi i timerjobbet, og sender via HTTPS til de forskellige firmaer. På denne måde er kodeord og brugernavn i sikret i opbevaring og transit. Et meget vigtigt element ved denne dekryptering er sikker opbevaring af nøglen, og her er der plads til forbedring. Denne nøgle skal gemmes sikkert, og skal gemmes et sted, hvor kun timerjob brugeren har adgang til. Dette kunne gøres ved at oprette en timerjob bruger, med en hemmelig mappe, og forbyde alle andre adgang til denne mappe. Nøglen til dekryptering burde også ændres en gang i mellem.

#### Front-end

Det kan konkluderes at front-enden er forholdsvis sikker. Dette kan det da der er implementeret login og grupper, så man skal være logget ind for at kunne få adgang til bestemte sider på hjemmesiden. Hvis man ikke er logget ind og skrive f.eks. ”http://130.226.195.251/Admin/AddSensor” i browseren, vil man blive omdirigeret til login siden. Der er dog en mindre mangel, i form af at er man logget ind og er man ikke en administrator, vil man kunne tilgå førnævnte link ved at skrive den i browseren.

Gruppe funktionerne på hjemmesiden fungerer uden problemer, da man ikke på nogen måde kan få adgang til en sensor som ikke er en del af en af de gruppe, man er medlem af.

## Procesvurdering

Gruppen har været enige om at starte på projektet så tidligt som muligt, og arbejde regelmæssigt gennem hele projektperioden, for at kunne korrigere for uforudsete udfordringer eller forsinkelser tidligere i forløbet.

Vi startede ud med at have en milestoneplan, men da den skulle rettes til ved alle møder med ole blev den oprindelige ikke overholdt, hvorefter det var nødvendigt med en revurdering, af vores arbejdsmetode. Milestonplanen er beholdt, og fulgt så godt som muligt, men arbejdsmetoden har været mere agil.

Den agile arbejdsmetode har for os fungeret godt, i den forstand at vi har kunne arbejde tæt sammen med kunden. Med dette tætte arbejde har vi kunne afklare hurtigt om vi arbejdede i den rigtige retning af hvad kunden ønskede af slutprodukt.

Overordnet er projektet delt ud i store dele sådan at vi hver i sær havde vores fokusområde. Ved hvert møde er der blevet delt nye mindre opgaver ud som skulle være løst til næste møde. Samtidig har vi diskuteret løsningerne igennem for de enkelte små som store opgaver, sådan at alle har været med til at løse dem.

## Forbedringer

1. Implementering af en load balancer, så der er mulighed for at tilføje flere applikationsservere til systemet.
2. Implementere intrusion detection system på sereveren.
3. Kryptering af hele databasen.
4. Databasen er normaliseret på nær en tabel. Denne tabel er implementeret uden normalisering, og denne tabel skal normaliseres for at sikre en optimal datastruktur.
5. Krypteringsnøglen til timerjobbet og hjemmesiden skal gemmes på et sted på serveren, hvor kun disse to har adgang til den.
6. Gatewayen skal kunne håndtere nye API’er, som f.eks. et ny ThingSpeak eller IC-Meter.
7. Der skal være muligt at kunne sammenligne to grafer på hjemmesiden, uden at skrive kode i Zeppelin.
8. Det skal ikke være muligt for en almindelig bruger at kunne tilgå administrationssiderne, ved at indtaste URL’en i browseren.

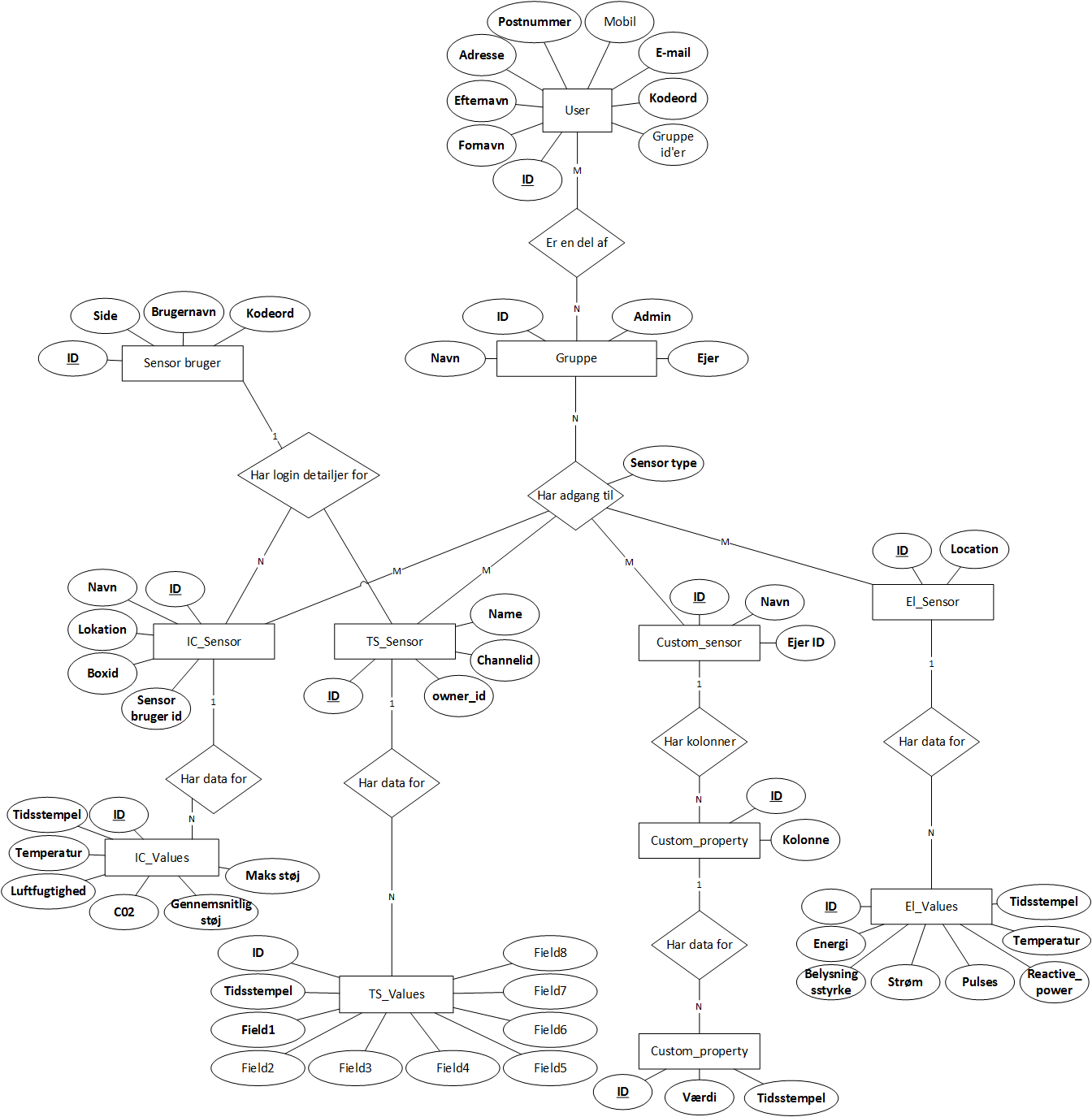
# Appendiks

## Arbejdsfordeling

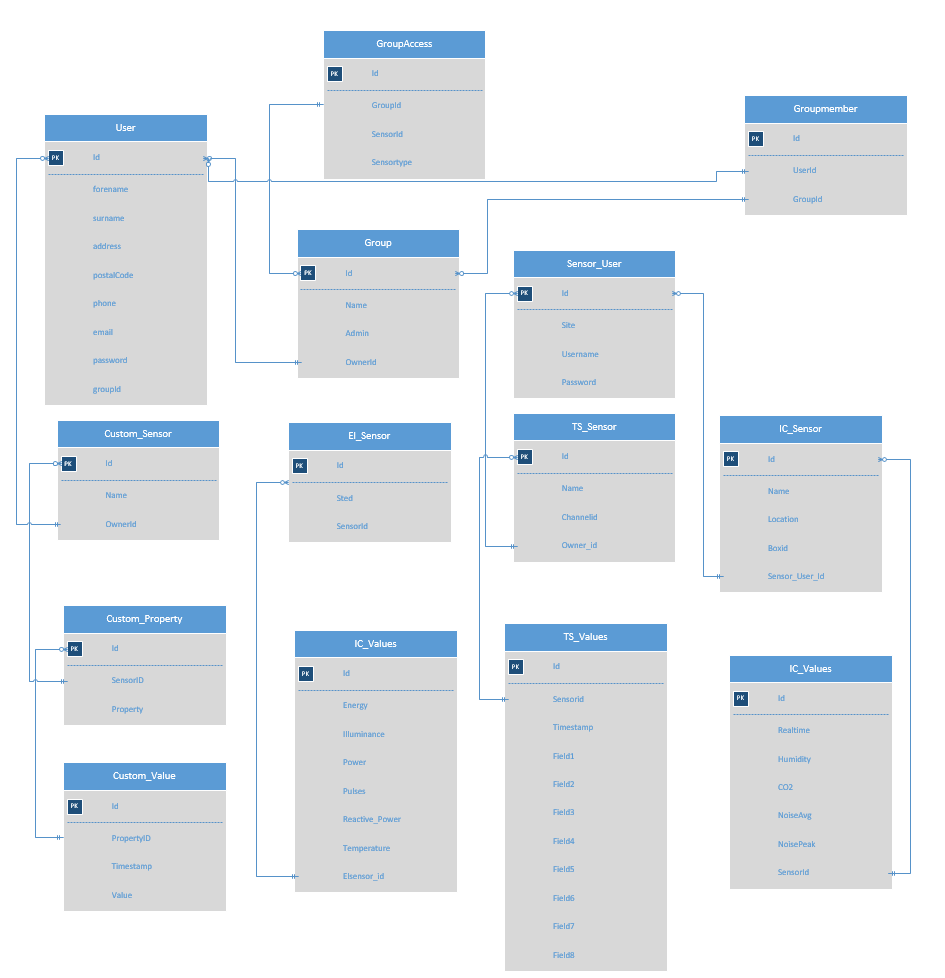
Tabellen nedenfor viser hvem der har været ansvarlig, for de enkelte dele af projektet. Vi har alle sammen været inde over hele projektet, og har arbejdet med hinandens arbejdsområder.

|  |  |
| --- | --- |
| Arbejdsområde | Ansvarlig |
| Server | Abdi |
| Database | Hakan |
| Timerjob | Hakan |
| API | Abdi, Nicklas |
| Front-end | Nicklas |
| Databehandling | Abdi |

## Database ER-diagram



## RID



## Dataanalyse

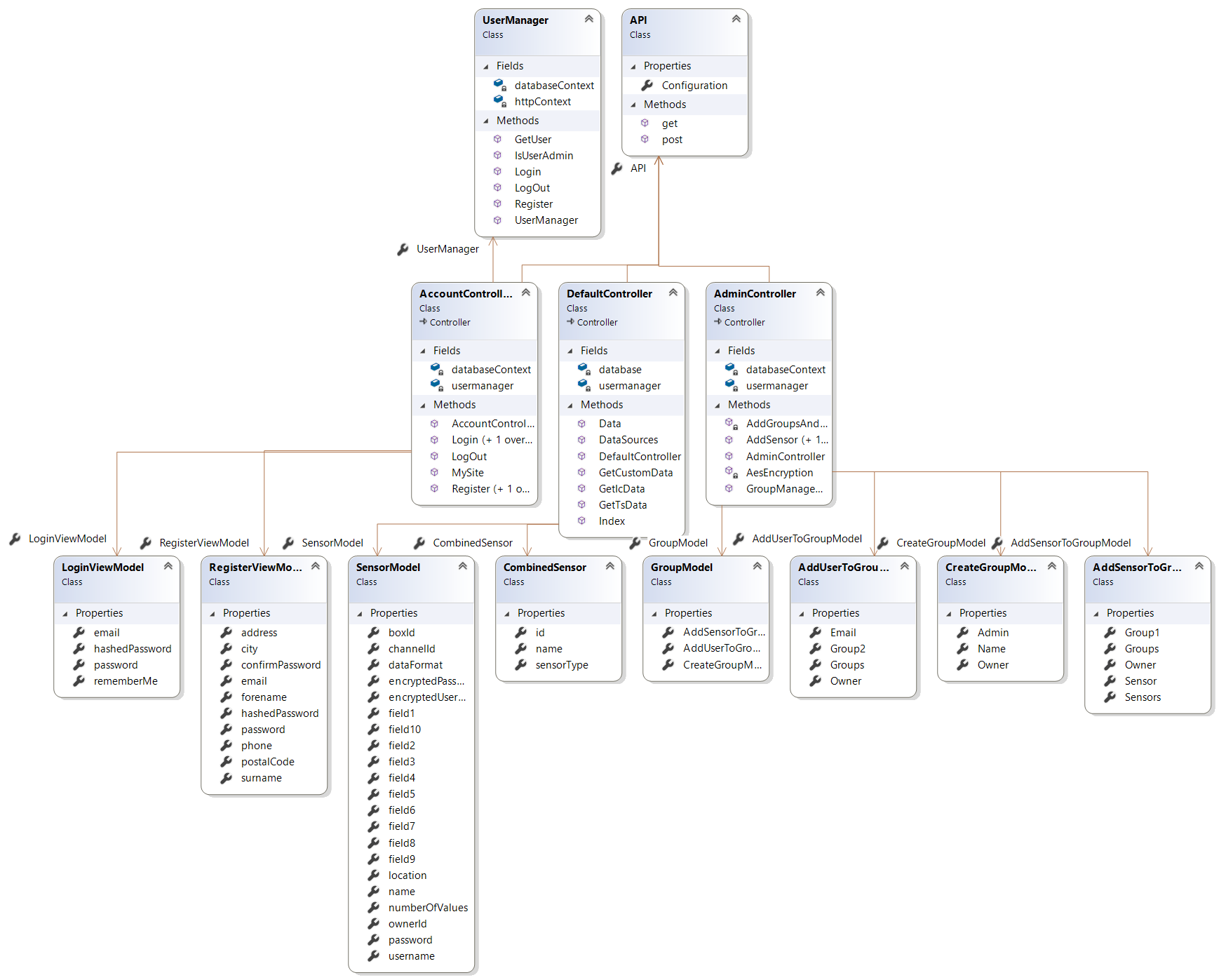
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Modtage / Sende** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Samling interval | Hver 5 min. |  |  |  |  |  |
| Pakkestørrelse | 700 bytes |  |  |  |  |  |
| Antal målere | 10 |  |  |  |  |  |
| Værdier målt | 4 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 7 KB | 84 KB | 2.016 MB | 14,11 MB | 60,48 MB | 725,75 MB | 2,18 GB |
| 10 rækker | 120 rækker | 2.880 rækker | 20.160 rækker | 86.400 rækker | 1.036.800 rækker | 3.110.400 rækker |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Samling interval | Hver 5 min. |  |  |  |  |  |
| Pakkestørrelse | 285 bytes |  |  |  |  |  |
| Antal målere | 100 |  |  |  |  |  |
| Værdier målt | 4 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 70 KB | 840 KB | 20.2 MB | 141,1 MB | 604,8 MB | 7,26 GB | 21,8 GB |
| 100 rækker | 1200 rækker | 28.800 rækker | 201.600 rækker | 864.000 rækker | 10.368.000 rækker | 31.104.000 rækker |
|  |  |  |  |  |  |  |
| Samling interval | Hver 5 min. |  |  |  |  |  |
| Pakkestørrelse | 1500 bytes |  |  |  |  |  |
| Antal målere | 100 |  |  |  |  |  |
| Værdier målt | 10 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 150 KB | 1,8 MB | 43,2 MB | 302,4 MB | 1,30 GB | 15.55 GB | 46,66 GB |
| 100 rækker | 1200 rækker | 28.800 rækker | 201.600 rækker | 864.000 rækker | 10.368.000 rækker | 31.104.000 rækker |
|  |  |  |  |  |  |  |
| Samling interval | Hver 20 min. |  |  |  |  |  |
| Pakkestørrelse | 700 bytes |  |  |  |  |  |
| Antal målere | 10 |  |  |  |  |  |
| Værdier målt | 4 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 7 KB | 21 KB | 504 KB | 3.53 MB | 15,12 MB | 181,44 MB | 544,32 MB |
| 10 rækker | 120 rækker | 2.880 rækker | 20.160 rækker | 86.400 rækker | 1.036.800 rækker | 3.110.400 rækker |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Lagring** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Samling interval | Hver 5 min. |  |  |  |  |  |
| Størrelse af én værdi på server | 6 bytes |  |  |  |  |  |
| Antal målere | 10 |  |  |  |  |  |
| Værdier målt | 4 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 240 bytes | 2,88 KB | 69,12 KB | 483,84 KB | 2,07 MB | 24,89 MB | 74,65 MB |
| 10 rækker | 120 rækker | 2.880 rækker | 20.160 rækker | 86.400 rækker | 1.036.800 rækker | 3.110.400 rækker |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Samling interval | Hver 5 min. |  |  |  |  |  |
| Størrelse af én værdi på server | 6 byts |  |  |  |  |  |
| Antal målere | 100 |  |  |  |  |  |
| Værdier målt | 4 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 2,4 KB | 28,8 KB | 691,2 KB | 4,84 MB | 20,7 MB | 248,9 MB | 746,5 MB |
| 100 rækker | 1200 rækker | 28.800 rækker | 201.600 rækker | 864.000 rækker | 10.368.000 rækker | 31.104.000 rækker |
|  |  |  |  |  |  |  |
| Samling interval | Hver 5 min. |  |  |  |  |  |
| Størrelse af én værdi på server | 6 bytes |  |  |  |  |  |
| Antal målere | 100 |  |  |  |  |  |
| Værdier målt | 10 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 6 KB | 72 KB | 1,73 MB | 12,10 MB | 51,9 MB | 622,8 MB | 1,87 GB |
| 100 rækker | 1200 rækker | 28.800 rækker | 201.600 rækker | 864.000 rækker | 10.368.000 rækker | 31.104.000 rækker |
|  |  |  |  |  |  |  |
| Samling interval | Hver 20 min. |  |  |  |  |  |
| Størrelse af én værdi på server | 6 byts |  |  |  |  |  |
| Antal målere | 10 |  |  |  |  |  |
| Værdier målt | 4 |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 1 måling | 1 time | 1 døgn | 1 uge | 1 måned | 1 år | 3 år |
| 240 bytes | 720 bytes | 17,28 KB | 120,86 KB | 518,4 KB | 6,22 MB | 18,66 MB |
| 10 rækker | 120 rækker | 2.880 rækker | 20.160 rækker | 86.400 rækker | 1.036.800 rækker | 3.110.400 rækker |

## API klassediagram

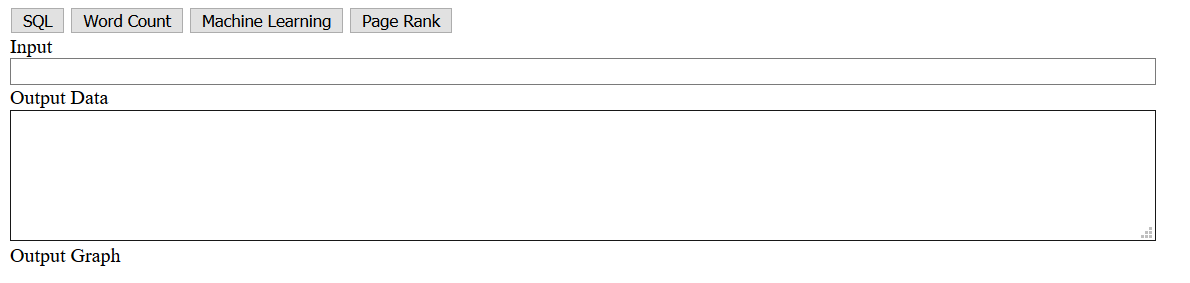


## Front-end klassediagram



## Forsøg på egen platform

Vi vil implementerer vores egen databehandling platform ind i hjemmeside.Herved vil der gennem vores egen platform, kunne overføres data fra databasen samt lave databehandling med Spark, således dataene kan visualiseres. De to andre platform, der er analyseret er vurderet til at kræve meget teknisk viden før systemerne kan benyttes. Af den grund har der været fokus på at gøre det så brugervenlig som overhovedet muligt, da der var en forestilling om, at det skulle foregå på den måde at funktionen på de enkelte sensorer aktiveres, hvorved den bagvedliggende kode visualisere og databehandler det givne data.

Vores egen data platform er blevet udviklet i Python, hvor det benyttes til den eksisterende webapplikation som er oprettet i ASP.net. API`et har tre metoder til at lave databehandling til en given datasæt.  

Figur 60: Illustration af hvordan det ser ud på hjemmesiden

.

Til implementering af vores egen platform er der taget udgangspunkt i Apache Python Spark (PySpark). Programmet er udviklet i Python, hvor vi vil lave et API som indeholder metoderne.

Apache Spark Spark Ecosystem indeholder følgende features, som bruges til at udvikle vores platform:

* **Spark Streaming**

Spark Streaming anvendes til processer som real-time streaming data, ved at anvende DStram der består af flere RDD. Det medfører at, det muligt og processerer real time data. Et alternativ kunne være og implementerer det med Firebase, så vi kan stream alt data i real-time.

* **Spark SQL**

Spark SQL kan vha. JDCB køre SQL queries på Spark datasættet. Desuden er Spark SQL grundlaget for, at brugeren kan overføre data fra forskellige dataformater. Herved gøres der brug af DStream, som består af flere RDD. Det gøre, det muligt at processere real-time data. Her kunne det have været muligt og implementer det med Firebase.

Spark SQL kan ved hjælp af JDCB tillade at køre SQL queries på Spark datasæt. Desuden danner denne Spark SQL grundlaget for, hvorledes brugeren kan overføre data fra forskellige dataformater, herunder JSON. På denne måde kan det transformere data, som er brugt til Spark SQL queries til at hente data fra MySQL

Efter og have udført metoderne i Python, benytter vi JavaScriptet til at illustrerer data visuelt.

|  |  |  |
| --- | --- | --- |
| **Request type** | **Metodenavn** | **Forklaring** |
| **GET** | SparkAPI/Spark\_mysql | Få data fra databasen via spark |
| **GET** | SparkAPI/Spark\_wordcount | Wordcount |
| **GET** | SparkAPI/Machinelearning | Machine learning |

### Delkonklusion

På baggrund af analysen konkluderes, at den selvvalgt platform krævede en del ressourcer som vi ikke havde kendskab til eller adgang til de krævet materialer. Dette skyldes, hvorledes vi i gruppen var nødt til at søge efter andre alternativer, som gjorde det mere besværligt. Zeppelin var et færdigt udviklet program. Af den grund valgte vi at tage udgangspunkt i den, da den indeholdte de ønsket funktioner.

1. <http://www.cmswire.com/cms/industry-news/-mono-runs-net-on-linux-000879.php> [↑](#footnote-ref-1)
2. <http://www.jqwidgets.com/license/> [↑](#footnote-ref-2)
3. <https://spark.apache.org/sql/> [↑](#footnote-ref-3)
4. <https://zeppelin.apache.org/download.html> [↑](#footnote-ref-4)