# Fitness Junkies I/S



Lærer: Henrik Thomsen

Elever: Tobias Brammer, Steffen Thomsen og Jonas Grendslev

Index

[Fitness Junkies I/S 1](#_Toc536779513)

[Titelblad 5](#_Toc536779514)

[Estimeret tidsplan 6](#_Toc536779515)

[Realiseret tidsplan 8](#_Toc536779516)

[Jonas Dagbog: 9](#_Toc536779517)

[Tobias Dagbog: 10](#_Toc536779518)

[Steffen Dagbog 11](#_Toc536779519)

[Læsevejledning 12](#_Toc536779520)

[Begrundelse for teknologivalg: 13](#_Toc536779521)

[Backend: 13](#_Toc536779522)

[Behov 13](#_Toc536779523)

[Restful web API 13](#_Toc536779524)

[NodeJs 14](#_Toc536779525)

[ExpressJS 15](#_Toc536779526)

[MySql - queries 17](#_Toc536779527)

[Authentication - Passport 19](#_Toc536779528)

[Passport basic - email 19](#_Toc536779529)

[Passport facebook 20](#_Toc536779530)

[Profilbillede - Base64 string 22](#_Toc536779531)

[Cookies: Redis session store 22](#_Toc536779532)

[Password politik 24](#_Toc536779533)

[Bcrypt: Password hashing 25](#_Toc536779534)

[MySql server 28](#_Toc536779535)

[Design 30](#_Toc536779536)

[Foreslå matches - Trigger 31](#_Toc536779537)

[Docker 32](#_Toc536779538)

[Docker compose 33](#_Toc536779539)

[Frontend: 34](#_Toc536779540)

[Swift: 34](#_Toc536779541)

[cocoapods: 34](#_Toc536779542)

[Facebook sdk: 35](#_Toc536779543)

[AvatarImageView: 37](#_Toc536779544)

[URLSession: 38](#_Toc536779545)

[Location tracking: 39](#_Toc536779546)

[opsætning af tracking af en route: 40](#_Toc536779547)

[UnitConverter extention(apple) : 40](#_Toc536779548)

[FormatDisplay: 42](#_Toc536779549)

[class User: Decodable: 43](#_Toc536779550)

[Profilbillede base64: 44](#_Toc536779551)

[Requestes: 46](#_Toc536779552)

[Opstart af Fitness Junkies netværk 47](#_Toc536779553)

[Design: 47](#_Toc536779554)

[Valg af udstyr: 50](#_Toc536779555)

[Topologi(Topologi vedhæftet i bilag, så man kan zoom): 51](#_Toc536779556)

[IP Plan: 52](#_Toc536779557)

[Opsætning af netværk 53](#_Toc536779558)

[Subnets: 53](#_Toc536779559)

[Netværks konfiguration: 53](#_Toc536779560)

[Netværksprotokoller: 54](#_Toc536779561)

[FHRP: 54](#_Toc536779562)

[HSRP: 54](#_Toc536779563)

[Preempt featuren: 55](#_Toc536779564)

[Track featuren: 55](#_Toc536779565)

[Hello timers: 55](#_Toc536779566)

[HSRP Versions: 55](#_Toc536779567)

[Konfiguration: 56](#_Toc536779568)

[VRRP: 57](#_Toc536779569)

[GLBP: 57](#_Toc536779570)

[Port security: 59](#_Toc536779571)

[Maximum MAC: 59](#_Toc536779572)

[Portfast: 59](#_Toc536779573)

[Storm-Control: 60](#_Toc536779574)

[DMZ: 61](#_Toc536779575)

[Opsætning: 61](#_Toc536779576)

[Opstart af fitness Junkies Server Drift 63](#_Toc536779577)

[Server topologi(Topologi vedhæft i bilag så man kan zoom): 63](#_Toc536779578)

[Begrænsning: 63](#_Toc536779579)

[Valg af udstyr 63](#_Toc536779580)

[Software, Services samt valg og opsætning af dem: 64](#_Toc536779581)

[Windows Server 2016 Datacenter: 64](#_Toc536779582)

[Installation: 64](#_Toc536779583)

[NIC Teaming Opsætning: 64](#_Toc536779584)

[Opsætning af Hyper-v 65](#_Toc536779585)

[Opsætning af Domain Controllere: 67](#_Toc536779586)

[Fejlfinding: 68](#_Toc536779587)

[Opsætning af Windows Cluster: 68](#_Toc536779588)

[DFS: 71](#_Toc536779589)

[DFS på Fileshare: 72](#_Toc536779590)

[Backup: 73](#_Toc536779591)

[Opsætning af DMZ Zone og Web Server: 73](#_Toc536779592)

[SNMP: 75](#_Toc536779593)

[Opsætning af Monitoring Server: 76](#_Toc536779594)

## Titelblad

Uddannelse: Datatekniker

Hold: HOT8\_DT6\_Programmering

HOT8\_DT6\_Infrastruktur

Emne: Motionsfællesskab

Elever: Tobias Brammer

Jonas Granslev

Steffen Thomsen

Eksamensperiode 07-01-2019 til 08-02-2019

Dokument afleveret 01-02-2019

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tobias Brammer

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Jonas Granslev Steffen Thomsen

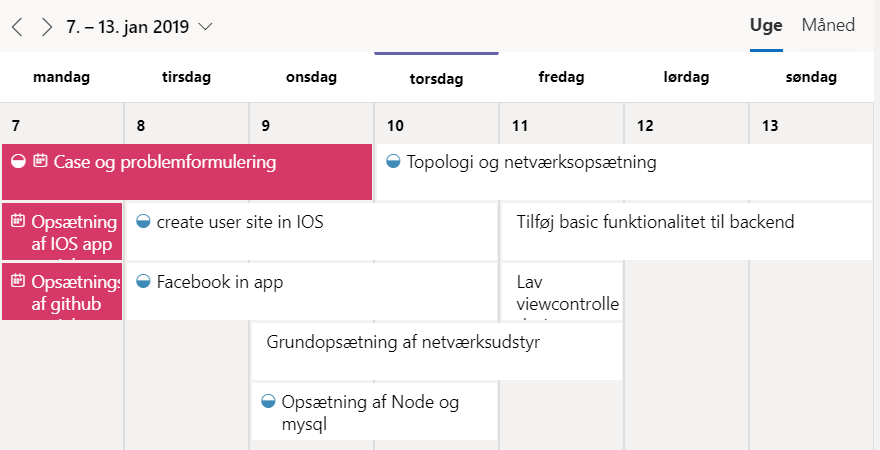
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

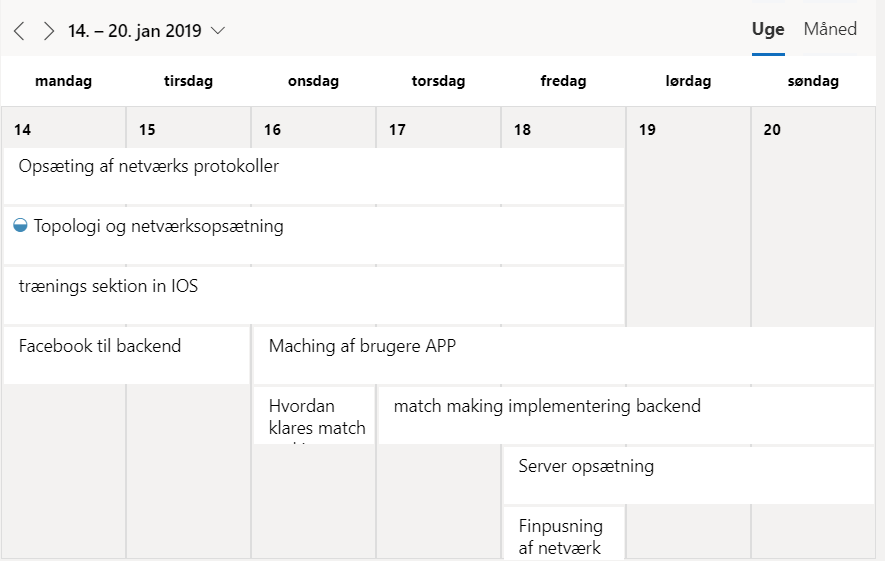
Henrik Thomsen

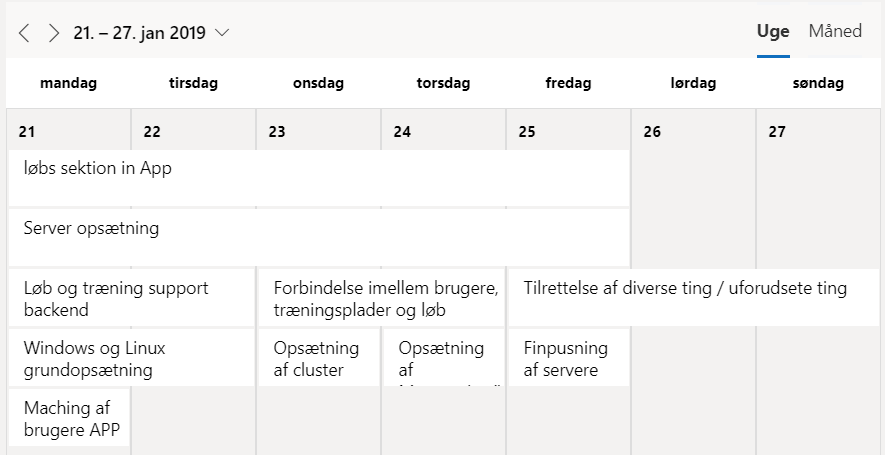
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Censor Censor

## Estimeret tidsplan





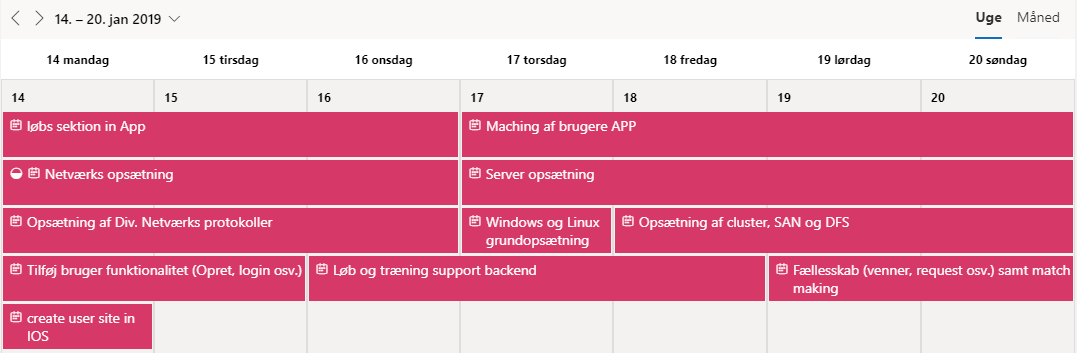






## Realiseret tidsplan











## Jonas Dagbog:

07-01-2019: Opstart, skrev case og problemformulering

08-01-2019: Case og problemformulering

09-01-2019: Case, problemformulering, finpudsning og godkendelse.

10-01-2019: Begyndte på netværksopsætning. Startede med at lave en topologi tegning over netværket i fitness junkies, og begyndte så småt på at stille det fysiske udstyr op.

11-01-2019: Konfigureret basic konfiguration på netværksudstyr og sørge for der var hul igennem netværket.

13-01-2019: Rapport skrivning

14-01-2019: Opsætning af diverse routing protokoller.

15-01-2019: Fortsatte med opsætning af routing protokoller.

16-01-2019: Fortsatte med opsætning af routing protokoller.

17-01-2019: Idag begyndte jeg at sætte udstyr op til servere og basic konfigurere vores windows og linux maskiner

18-01-2019: Begyndte på opsætning af cluster, SAN og DFS miljø

20-01-2019: Rapport skrivning

21-01-2019: Fortsatte med opsætning af cluster, SAN og DFS miljø

22-01-2019: Fortsatte med opsætning af cluster, SAN og DFS miljø

23-01-2019: Opsætning af Ubuntu 18 Server. Monitoring server og Web server i DMZ zone.

24-01-2019: Finpusning af Server miljø

25-01-2019: Finpusning af netværk

26-01-2019: Rapport skrivning

27-01-2019: Rapport skrivning

28-01-2019: Rapport skrivning

29-01-2019: Rapport skrivning

30-01-2019: Rapport skrivning

31-01-2019: Rapport skrivning

01-01-2019: Rapport skrivning

02-01-2019: Forberedelse til svendeprøve

03-01-2019: Forberedelse til svendeprøve

04-01-2019: Forberedelse til svendeprøve

05-01-2019: Forberedelse til svendeprøve

06-01-2019: Forberedelse til svendeprøve

## Tobias Dagbog:

07-01-2019: Skrev Case og problemformulering

07-01-2019: Lavede Step Signup System for IOS app

08-01-2019: Facebook i app (uden sammenspil med api'et)

09-01-2019: Arbejde med oprettelse og login af bruger, både ui og funktioner.

10-01-2019: Fortsat arbejde med login og oprettelse af bruger.

11-01-2019: Fortsat arbejde med login og oprettelse af bruger.

12-01-2019: Fortsat arbejde med login og oprettelse af bruger.

13-01-2019: Fortsat arbejde med login og oprettelse af bruger.

14-01-2019: arbejde med løbs sektionen og tracking af bruger iforhold til at oprette et løb med rute.

15-01-2019: gem af rute og mapView med polyline af route.

16-01-2019: get previous runs og tilføj venner til din løbetur.

17-01-2019: Matching af brugere ui

18-01-2019: Fortsæt med matching af brugere med fokus på detail view.

19-01-2019: Fortsæt med matching af brugere, med fokus på friend request

20-01-2019: arbejde med Fællesskab delen, med fokus på user profile site and profile picture implementation i hele appen.

21-01-2019: Fortsæt med fællesskabs delen.

22-01-2019: sidste del af fællesskabs delen med fejlrettelser og forbedringer.

23-01-2019: Begyndt arbejder på Fitness sektion af appen, med fokus på opret af en træningsplan.

24-01-2019: Fortsat arbejde på Fitness sektion af appen, med fokus på det sidste del.

25-01-2019: Fejl rettelser og facebook samspil med apiet.

26-01-2019: Rapport skrivning

27-01-2019: Rapport skrivning

28-01-2019: Rapport skrivning

29-01-2019: Rapport skrivning

30-01-2019: Rapport skrivning

31-01-2019: Rapport skrivning

01-01-2019: Rapport skrivning

02-01-2019: Forberedelse til svendeprøve

03-01-2019: Forberedelse til svendeprøve

04-01-2019: Forberedelse til svendeprøve

05-01-2019: Forberedelse til svendeprøve

06-01-2019: Forberedelse til svendeprøve

## Steffen Dagbog

07-01-2019: Oprettelse af Github repository samt arbejde med case og problemformulering.

08-01-2019: Arbejde med case og problemformulering.

09-01-2019: Færdiggørelse af problemformulering, samt start på opsætning af Express JS i node.js og design af bruger databasen.

10-01-2019: Fortsat arbejde med opsætning af Express JS i node.js og design af bruger databasen.

11-01-2019: Begynde at arbejde med web API. Oprettelse af bruger med email og password implementeret.

12-01-2019: Arbejde med login samt session til at huske brugeren.

13-01-2019: Arbejde videre med sessioner. Oprettelse af redis database til sessioner.

14-01-2019: Færdiggørelse af sessioner. Bruger sættes ind i sessionen og derefter sendes der en cookie til brugeren for at referere til sessionen med brugerens informationer.

15-01-2019: Opret bruger med alle informationer, altså navn, alder, location, profilbillede osv.

16-01-2019: Start på løb sektion(Gem, se og del løb)

17-01-2019: Design af database ift at en bruger skal tilknyttet et specifikt løb. Derudover færdiggørelse af gem og se løb.

18-01-2019: Et løb laves sådan at det kan deles med en bruger ud fra brugerens id. Dette gøres når løber registreres. Derved færdiggøres løb sektionen.

19-01-2019: Start på fællesskabsdelen med venner, requests osv.

20-01-2019: Send friend request laves.

21-01-2019: Funktioner til at få friend request, acceptere requests, afvise requests samt få venner.

22-01-2019: Fællesskabsdelen laves færdig, altså få liste af venner og søg efter venner. Derudover laves det sådan at man får ens foreslået venner.

23-01-2019: Der arbejdes videre med brugerfunktionalitet. Der tilføjes at brugere kan ændre deres informationer, password osv.

24-01-2019: færdiggørelse af udvidet brugerfunktionalitet.

25-01-2019 - 27-01-2019: Fejlrettelser, heriblandt fejl med profilbillede, småfejl i dokumentation samt fejl i asynkron kode.

# Læsevejledning

Til den IT ansvarlige i Fitness Junkies I/S med teknisk viden.

I Fitness Junkies I/S tager vi udgangspunkt i en proof of concept løsning, Hvilket vil betyde at nogle af de løsninger som er beskrevet, ikke er implementeret pga af manglende ressourcer. De vil dog være beskrevet her.

**Rapporten vil være opdelt i 4 områder bestående af:**

* Backend Steffen Thomsen
* Frontend Tobias Brammer
* Netværk Jonas Grendslev
* Server Jonas Grendslev

Henvis til Indholdsfortegnelsen for afsnit

# 

# Begrundelse for teknologivalg:

## Backend:

### Behov

Backend skal bestå af et web api, som kan snakke med flere forskellige operativsystemer. I første omgang er det bestemt at produktet skal understøtte IOS og Android.

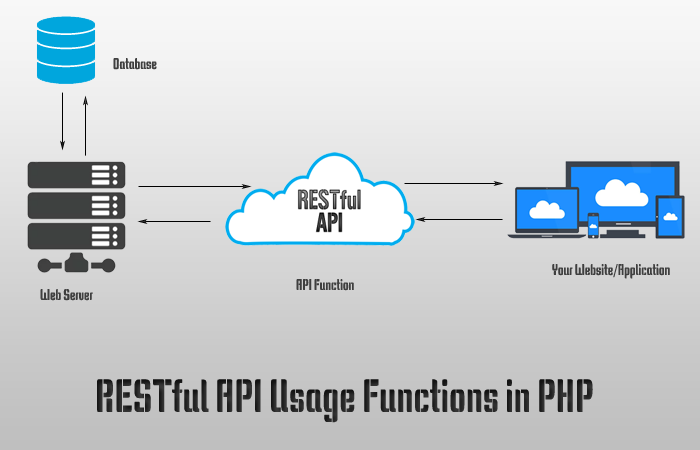
Derover skal der laves en database, hvor alle bruger-afhængige informationer gemmes, samt en database som holder styr på brugerens session/cookies. Når der kigges på hvordan brugerne skal oprettes, skal der både kunne gøres brug af almindeligt oprettelse af brugere via email, samt oprettelse via Facebook.

Det hele skal kunne køres op nemt, og derfor er en container teknologi valgt til dette.

### Restful web API

Til løsningen er der valgt at bruge RESTful API, som er brugt til at levere informationer til frontend. SOAP var også en kandidat, men da RESTful API er hurtigere, mere simpelt setup samt muligt at bruge på stort set alle enheder med forbindelse til internettet, faldt valget på RESTful API.

REST er en arkitektur der bygger på eksisterende systemer og HTTP protokollen, og er derfor genkendelig at bruge for alle der kender HTTP protokollen på forhånd. Dette kan bl.a. ses ved statuskoder, som er de samme som i HTTP. Derudover bruges SSL og TLS som kryptering.

RESTful api bruger derfor selvfølgelig standard HTTP requests. De mest brugte er GET, PUT, POST og DELETE.

GET - Få data

PUT - Ret data.

POST - opret ny data

DELETE - slet data

kilde:<http://www.kvcodes.com/medias/uploads/2017/02/RESTful-API-Usage-Functions-in-PHP.png>

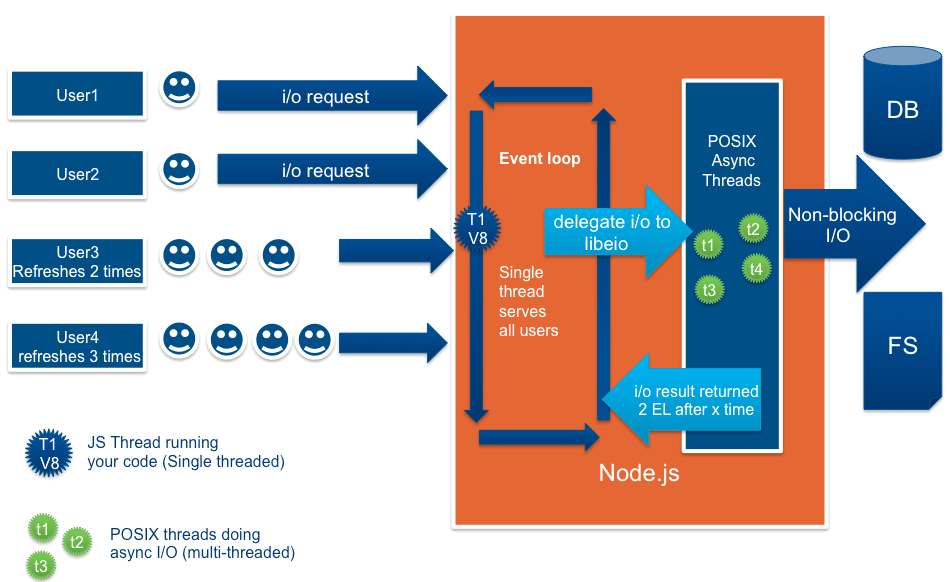
### NodeJs

NodeJs er bygget på JavaScript, og er derudover open source og multi platform(Windows, Linux, Unix, Mac OS osv.). NodeJS har også mange moduler, som især kan bruges til webudviklinger og derudover har vi tidligere erfaring med NodeJS brugt som web API.

NodeJS er også asynkron, hvilket betyder at ventetid er væk, og derfor er meget velegnet til API, da mange brugere skal have fat i APIét på samme tid.

Ved brug af bl.a. PHP vil der opstå ventetid når f.eks. en fil skal læses fra filsystemet. Her springer NodeJS blot videre og vender derefter tilbage når filen er klar.

NodeJS kører i en tråd, som er et event loop, hvor alle request bliver taget imod. Hvis requesten har brug for DB eller fil ressourcer, bliver de sendt videre til en I/O worker, som sørger for at det info der er brug for bliver hentet. Derfor kan event loopet fortsætte med at tage imod request, indtil at workeren har fået fat i den data der skulle hentes. Når event loppet så har tid, bliver dataene fra workeren givet videre til event loopet og der kan arbejdes videre med dataene.



[https://cdn-images-1.medium.com/max/1200/1\*LgvhkHtSWpv4IVx8jgc7Ag.png](https://cdn-images-1.medium.com/max/1200/1*LgvhkHtSWpv4IVx8jgc7Ag.png)

#### 

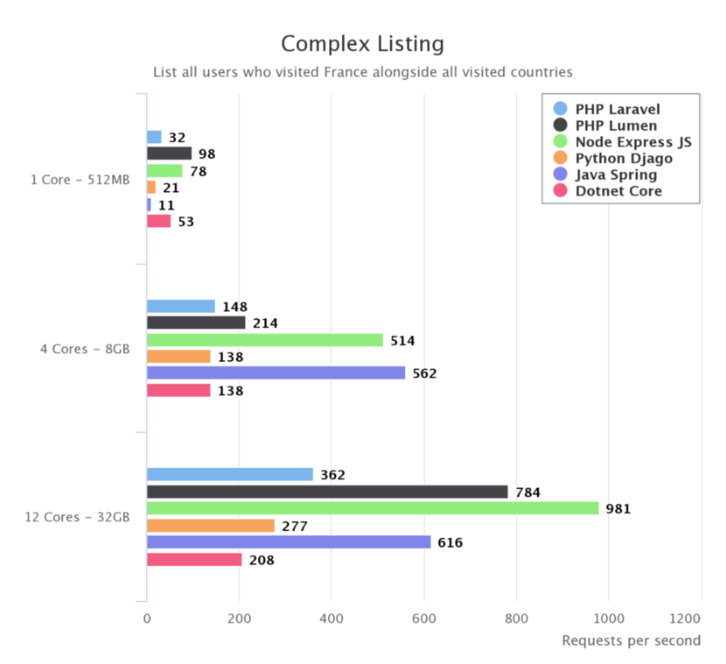
#### 

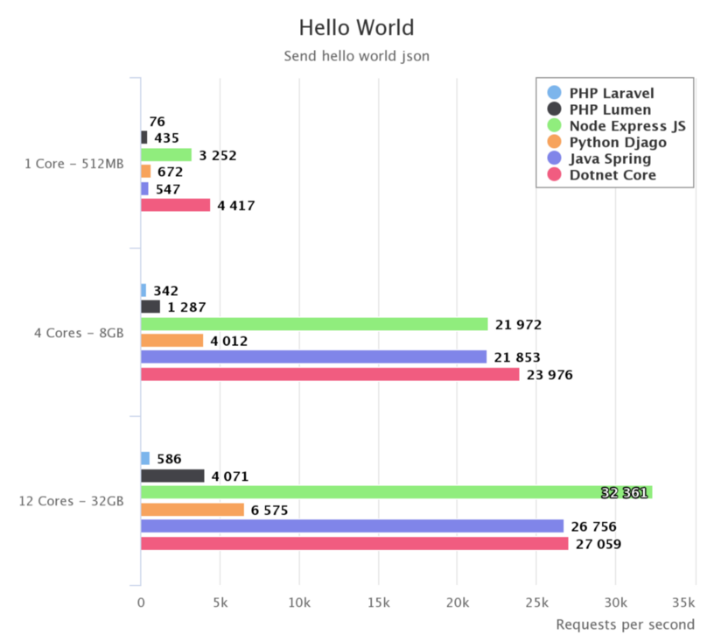
### ExpressJS

ExpressJS er et web framework til NodeJS, som er meget populært til at bygge web applikationer og APIér med, fordi det er meget hurtigt og er baseret på HTTP funktionalitet.

Vores valg af web framework er baseret på performance, simple opsætning samt mulighed for at gøre brug af databaser, sessioner, kryptering samt Facebook login på en simpel måde.

Når man kigger på performance, eller request på sekund, ved ExpressJS, ligger den som en af de hurtigst blandt de populære frameworks at bruge til web api.





<https://medium.com/@mihaigeorge.c/web-rest-api-benchmark-on-a-real-life-application-ebb743a5d7a3>

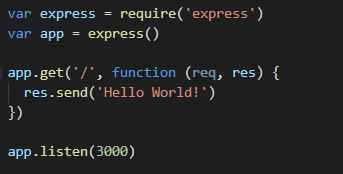
Det ses samtidig at express JS klarer sig bedre og bedre jo flere ressourcer der bliver tildelt, og idet at dette setup skal kører som et stort system (mange brugere) på længere sigt, vil ekspress JS være en rigtige løsning i forhold til systemets størrelse.

Brugervenligheden, altså hvor god dokumentation og hvor nemt det er at komme igang, er også et stort plus ved express JS. Den basale funktionalitet, altså routing og middlewares, er beskrevet helt fra nybegynder til de mere advanceret funktioner, såsom sessioner.

Routing i Express JS kan kort forklares ved at den ligger sig meget tæt op af HTTP protokollen og bruger derfor mange af de samme termer som man kender fra HTTP.

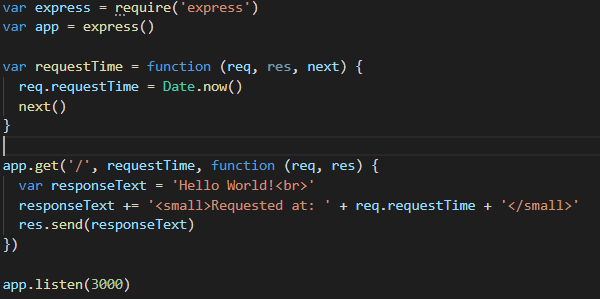
Middlewares kan kort forklares som de funktioner et kald består af. Middlewares kan tilgå request og svare på requesten, og derfor kan det ikke garanteres at alle middlewares for et kald bliver kørt.

Herunder ses den mest simple form for et HTTP kald der gør brug af en middleware funktion ved hjælp af Express JS:



app er i dette tilfælde et objekt som bliver brugt til at oprette routing ude fra Expres modulet. Der oprettes en webservice der lytter på port 3000 (nederste linje) og en GET request (app.get), som kaldes ved blot at tilgå domaine navnet (uden sti) som signaleres ved ***‘/’***. Kaldets middleware er ***function(req,res)***, hvor req er selve requesten(kan indeholde forskellig data som brugeren sender med) og res er response. I dette tilfælde vil der sendes en besked tilbage med ***Hello World***, idet send i reponse elementet sættes til dette.

Som tidligere skrevet kan flere middlewares også optræde i et kald. Et eksempel er vist nedenunder:



Igen lyttes der på port 3000 og en GET request oprettes. Det er er anderledes er at der er 2 middlewares for dette kald, nemlig ***requestTime*** og ***function(req,res)***. Disse 2 er angivet efter ***app.get('/',***. Det ses at ***requestTime*** har et nyt parameter kaldet ***next. next*** bruges til at fortælle at næste middleware skal køres. Det der derfor vil ske i dette kald er at ***requestTime*** kører først og får tiden fra requesten(ved brug af req) og derefter sendes til ***function(req,res)*** hvor der svares med ***Hello World! Requested at:*** efterfulgt af tiden.

Hvis en middleware altid skal kaldes som det første, kan man også bruge app.use(***Middleware function navn****).* Dette er gjort i denne applikation for at tjekke bruger sessioner (forklares i afsnittet om cookies)

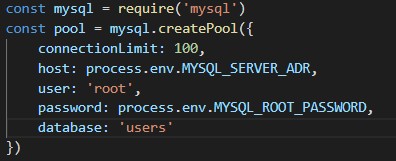
Dette forklare hvorfor Express JS er en oplagt kandidat set ud fra performance og simple basic funktionalitet. Derover understøtter Express JS også databaser(MySql), kryptering, session samt Facebook login, som vil forklare igennem de næste afsnit.

#### MySql - queries

Mysql bruges som databasen til bruger informationer, og derfor skal Express JS kunne skrive og tilgå disse data (MySql opsætningen er beskrevet i MySql server afsnittet)

Til dette bruges modulet ***mysql***, som gør det muligt at oprette forbindelse, samt sende queries til vores server.

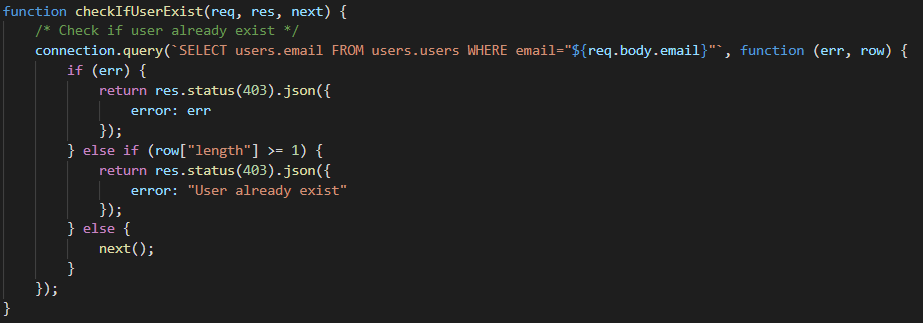
Nedenfor ses den opsætningen der skal til for at lave en forbindelse til MySql serveren, lavet som en pool af connections.



En pool af connectionen opretter en ny forbindelse hver gang der er brug for en. Dette kan ses som en worker der skal hente et resultat før der kan arbejdes videre med requesten. Når resultatet er hentet fra queryen lukkes forbindelses igen.

For at forbindelses kan oprettes som forventet skal host, user, password, samt database bruges. host og password ligger som ***process.env.MYSQL\_SERVER\_ADR*** og ***process.env.MYSQL\_ROOT\_PASSWORD***. Disse 2 er environment variabler, som sættes igennem Docker (container teknologi som forklares senere). Dette er gjort ved server adressen fordi den kan ændre sig og ved password fordi den ikke skal stå i klar tekst i koden.

Forbindelsen bruges derefter selvfølgelig til at sende queries afsted. Dette gøres ved at referere til pool objektet. Pool objektet ligger i sin egen fil, og eksporteres derfor for at kunne bruges af hvilken som helst anden fil i projektet. Ved import laves navnet om connection, da der kun laves en forbindelse når der refereres. Nedenfor ses et eksempel:



Dette er middleware funktionen ***CheckIfUserExist*** som kaldes når en bruger skal oprettes. Her ses ***connection.query*** brugt til at tjekke om nogle brugere allerede har den email en anden bruger forsøger at oprette sig med. Resultatet hentes ved hjælp af en callback til ***connection.query***, nemlig ***function(err, row)***. Her vil ***err*** blive sat hvis en fejl skete da dataene skulle hentet. Hvis ingen fejl skete bliver row sat til de data der passede til forespørgelse, og det kan tjekkes om brugeren allerede eksistere.

Som nævnt før er Node JS et asynkront sprog, og dette kan skabe problemer i forhold til den rækkefølge den svares til klienten, når det gælder databaser. Det er nemlig ikke tilladt at sende 2 svar til samme klient ud fra 1 request. Problemet opstår når der er 2 queries er i samme middleware samt callback. Her vil første query blive kaldt, men Node JS vil fortsætte til anden query med det samme. Hvis første query herefter bliver færdig med en fejl, som skal afsluttet kaldet med en fejl, vil Node JS fortsætte til anden query også (da denne allerede er i gang) og derfor også afslutte kaldet for anden gang, hvilket vil gøre at serveren går ned.

Heldigvis understøtter Node JS en løsning til dette, kaldet util.promisify.



util.promisify funktionen tager den callback som query har og laver det om til at returnere et promise, således at funktionen kan kaldes med await eller async. Her skal await bruges, for at Node JS venter med at gå videre til anden query indtil første query er helt færdig.

#### Authentication - Passport

Brugerne skal på en måde autentificere sig op mod APIét og til det er Passport JS brugt til login med email, samt oprettelse og login med Facebook.

Passport JS kan kort forklares som en middleware der kan tage sig af login fra mange forskellige tjeneste, heriblandt Facebook, Google, Twitter osv.

##### Passport basic - email

Under passport ligger et modul kaldet basic. Dette bruges til at logge ind med basic authentication. Denne type authentication bruges når der logges ind med en email-bruger. Informationen ligger i header og ser f.eks. således ud:



Dette indeholder email og password, som passport finder. Derefter tjekkes det af passport basic at der er en bruger hvor email og password matcher ud fra de modtagede data. Denne funktion ser således ud:

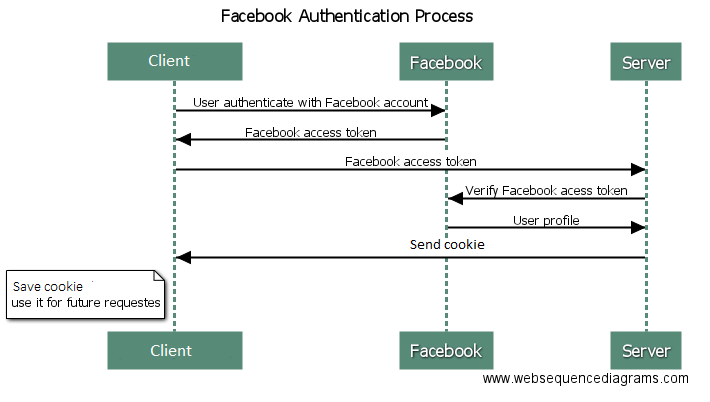


Alle brugere med den modtagede email hentes ud (Der kan kun være 1). Hvis en bruger er fundet, tjekkes det med bcrypt *(forklares i afsnittet om bcrypt)* om adgangskoden stemmer overens med den fundne bruger. Hvis adgangskoden matcher, returneres alt information om brugeren, for derefter at kan oprette en session *(forklares i afsnittet om cookies)* til brugeren.

##### Passport facebook

Passport modulet bruges også til Facebook oprettelse og login. Grunden til at det bruges ved begge requests, er at informationer kan hentes fra en Facebook konto, ved hjælp af en access token som modtages fra frontend.

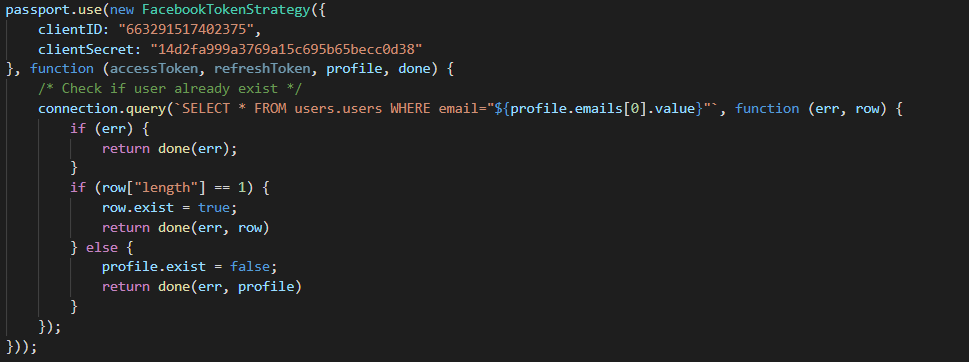
Hvis man kigger på hvordan opbygningen af brugen af Facebook i hele dette produkt (fra frontend til backend) er dette et godt diagram:



<https://codeburst.io/node-js-rest-api-facebook-login-121114ee04d8>

Her kan det ses at selve tilladelsen for at få adgang til brugerens Facebook bliver behandlet i klienten/frontend. Når tilladelsen er giver, fåes en access token, som sendes til backend. Herefter kan backend sende en request til Facebook for at få de ønskede informationer om brugeren. På den måde er det kun en access token der skal fra frontend til backend, hvilket ikke er særlig ressourcekrævende.

Implementering i passport ser således ud:

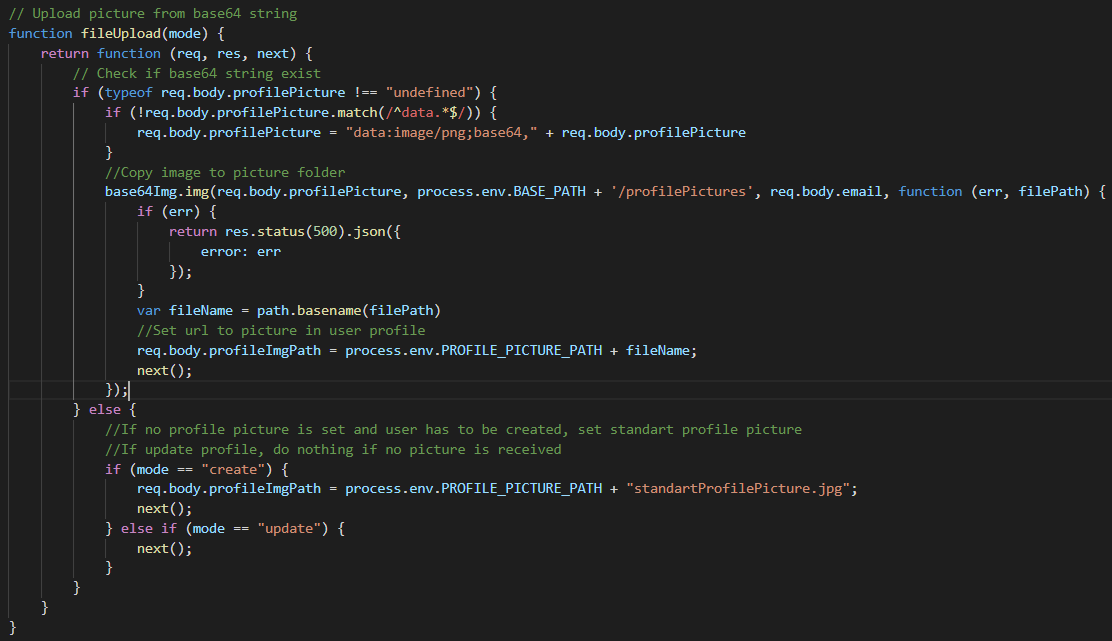


Her bruges en clientId og clientSecret. Disse informationer referer til en Facebook applikation, som skal oprettes for at der kan hentes informationer på brugere. Det er igennem denne Facebook applikation at brugeren giver lov til at informationerne kan bruges. På den måde kan brugeren altid holde styr på hvilke applikationer/virksomheder der har adgang til Facebook informationer og hvilke.

Informationerne fra facebook hentes og herefter kører funktionen ***function (accessToken, refreshToken, profile, done)***. I denne funktion tjekkes det om brugeren allerede findes. Ud fra om brugeren findes eller ej, sættes der et flag, så næste middleware ved om bruger allerede findes. Derudover sendes enter informationerne på den allerede oprettet bruger eller informationer fra Facebook med videre. Informationer fra Facebook skal bruges hvis en bruger skal oprettes, da navn og email tages fra Facebook profilen.

#### Profilbillede - Base64 string

Brugerne har mulighed for at oprette sig med og ændre profilbillede. Dette sker i en middleware funktion kaldet ***fileUpload***.



Middleware funktionen bruges både når en bruger opretter sig samt når et profilbillede skal opdateres.

Det tjekkes først om et profilbillede er sendt med. Hvis dette er tilfældet, gemmes billedet i mappen ***profilePictures***. Mappen position afhænger af hvor APIét køres fra, og derfor er den angivet som environment variablen ***process.env.BASE\_PATH***. Herefter sættes ***profileImgPath*** til den url hvor billedet kan hentes fra.

Hvis intet billede er sendt med ved oprettelse via email sættes profilbilledet til et standart profilbillede. Hvis der ikke er noget profilbillede sendt med når profilen skal opdateres, gøres der intet og man sendes direkte videre til næste middleware funktion.

#### Cookies: Redis session store

For at APIét skal kunne huske brugere når de er logget ind, bruges der cookies.

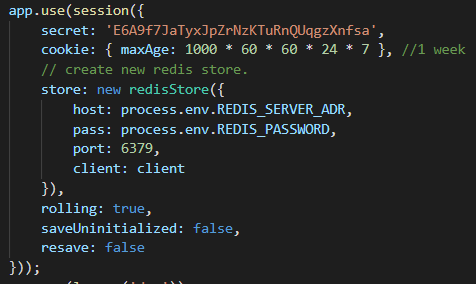
I denne applikation er dataene i cookies gemt i Redis, og er derfor server side cookies, også kaldet sessioner. Redis databasen kører op via Docker som beskrives senere i afsnittet om Docker.

Redis er applikationens session store, og det er derfor i Redis at alt information omkring sessioner ligger. I sessioner ligger bl.a. det meste information om brugeren, udløbstid og et unikt ID som sessionen har.

Når en klient logger ind eller opretter sig som bruger, vil en cookie blive sendt med til brugeren som indeholder ***connect.sid***. Cookien vil herefter bliver sendt med hver gang brugeren laver en request til APIet. Her vil ***connect.sid*** bliver brugt af APIet til at finde ud af om brugeren har en valid session, ved at slå den op i Redis. Dette har en række fordele:

* Brugerens request kan afvises med det samme hvis ingen gyldig session er fundet ud fra ***connect.sid*** (medmindre at opret bruger eller login er kaldt)
* APIét har automatisk data om hvilken bruger der har sendt requesten

I APIét sættes Redis til at blive brugt til sessioner:

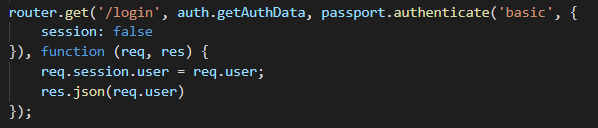


Der bruges her 6 elementer fra express-session biblioteket, nemlig secret, cookie, store, rolling, saveUninitialized og resave.

* ***Secret***: Det første element (E) vil blive brugt til at signe sessionen imens resten af stregen bruges når en session skal valideres.
* ***cookie***: Værdien maxAge i cookie objektet sættes til en standard værdi på 1 uge. Dette vil sige at hvis brugeren ikke bruger appen i en uge, skal der logges ind igen.
* ***store:*** Angiver hvilken session store der skal bruges, samt dens indstillinger.
* ***rolling***: Sørger for at cookien altid bliver sat til at udløbe om 1 uge efter en request. Derved udløber en cookie aldrig hvis appen bruges engang imellem.
* ***saveUninitialized***: Sættes til false for ikke at gemme sessioner når de ikke er initialiseret.
* ***resave***: Sættes til false for at sessioner kun skal ændre data i Redis når dataene ændre sig

i ***store*** elementet oprettes der forbindelses til selve Redis databasen. Ligesom ved MySql bruges der igen environment variabler til at sætte host og password, da host kan ændre sig og password ikke skal stå i plain tekst.

Herefter er Express JS klar til at gøre brug af sessioner. Sessioner sættes på følgende måde:



Dette er users/login requesten for at kunne logge ind med en email bruger. Brugeren bliver valideret af middleware funktionen ***passport.authenticate('basic',*** hvorefter middleware funktionen ***function(req, res)*** bliver kaldt. Her sætter ***req.session.user*** til et objekt af user som ligger i requesten(sat i passport). Cookien samt informationer om den bruger der logget ind, sendes derefter tilbage til brugeren.

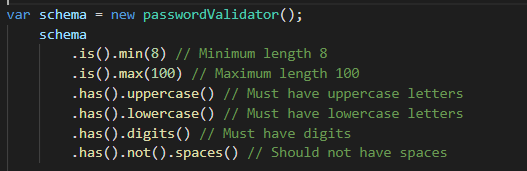
Hver gang brugeren sender en request, vil det blive tjekket om ***req.session.user*** har nogle data. Disse data hentes fra Redis, ved at tjekke på den cookie brugeren sender.

#### Password politik

I backend er der implementeret en password politik, for at være sikker på at brugeren laver et stærkt password, for på den måde at:

* Gøre det svære at gætte en brugers password.
* Gøre det svære at bruteforce en enkelt brugers password.
* Gøre det svære at dehashe passwords, hvis databasen bliver tilgået af folk med uretmæssig adgang.

Password politikken består af et sæt regler, som tjekkes hver gang en bruger vil oprette sig via email.



Ud fra dette sæt af regler vil der være krav om at karaktererne minimum skal være a-z, A-Z samt 0-9. Dette giver 62 forskellige muligheder ved hver karakter og ved at der minimum skal være 8 karaktere i et password vil der være:

62^8 = ***218340105584896 eller 2.18 \* 10^14***

forskellige kombinationsmuligheder. Dette gælder ved at der er 8 karaktere og stiger grædvist efter hvor mange karaktere et password består af. Dette kan ses ved f.eks tage med 16 karaktere:

62^16 = ***47672401706823533450263330816 eller 4.77 \* 10^28***

#### Bcrypt: Password hashing

Nu hvor der styr på password regler, skal der kigges på hvordan passwords beskyttes i en database. Men hvorfor skal passwords hashes når dette er en fitness applikation og hvad afgøre hvad der bedst at beskytte med?

Password skal bestykkes i alle applikationer, da det ellers kan have store konsekvenser hvis alle data bliver stjålet. Dette kan ske ved at brugerne har trang til at bruge samme email og password til mange forskellige services, da de ellers til sidst kan have så mange forskellige passwords, at de glemmer dem alle. Derved kan passwordet brugt i denne applikation også godt være brugt på langt vigtigere og mere personfølsomme sider, og derfor kommer brugerens digitale sikkerhed i fare. Umiddelbart virker dette som en brugerfejl, men for virksomhedens image kan det have store konsekvenser, da brugerne vil få mistillid til at deres data bestykkes som det forventes.

Når man kigger på hvordan password skal hashes, så tænkes der på hvor lang tid det vil tage at finde den streng der passer til hashen, og derved have det rigtige password. Samtidig må det ikke tage så lang til at finde en hash til et password at brugeren skal sidde og vente på det. Her er bcrypt et godt valg, da man kan definere hvor stærk en hash skal være. Her ses det hvor længe det vil tage at finde en hash til et password ud fra hvilken cost den er angivet:



<https://www.npmjs.com/package/bcrypt>

Dette tager udgangspunkt i en 2 GHz processor og det ses at hvis reglen om at brugeren ikke skal opfatte at passwords hashes, skal man helst bruge under 13 runder, da det ellers tager alt for lang tid.

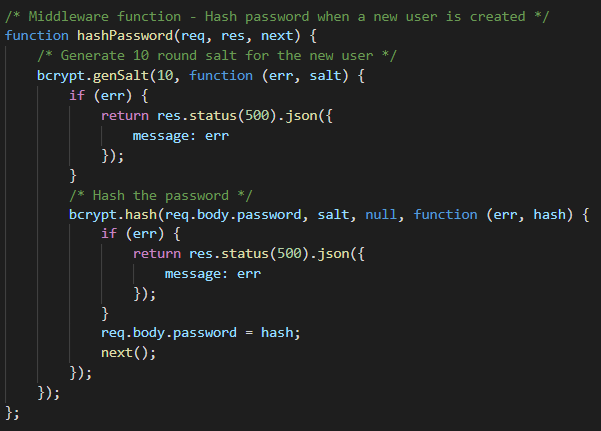
Til denne applikation er der valgt 10 runder som cost (omkring 100 ms), hvilket ikke lyder af særlig meget, men dette skal tænkes som den tid det tager at finde en hash til 1 streng.

Hvis man tager tallet for kombinationsmuligheder for 8 karaktere, altså ***218340105584896,*** og derefter ganger dette med 100 ms

***218340105584896 \* 100 ms = 21834010558489600 ms eller 691,9 år***

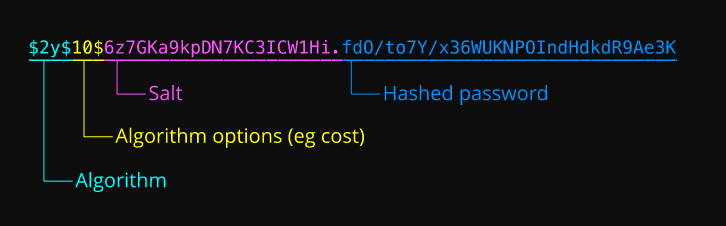
Derfor kan det ses at en cost på 10 er rigeligt. Resultatet er dog lavet med en 2 GHz processor, og hardware bliver selvfølgelig kraftigere, og derfor vil cost skulle hæves løbende.

Selve hashing af password sker i denne middleware funktion:



Først køres ***bcrypt.gensalt(10, function (err, salt)***, som generere en salt der skal bruges til sammen med passwordet for at få en hash. En salt generes ved hver password, for at hvis det samme password krypteres 2 gange, skal resultaterne være forskellige. På samme tid angives cost her, nemlig 10.

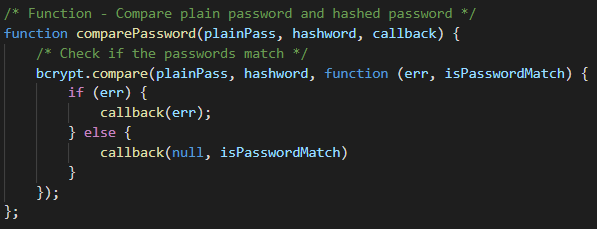
Herefter springes der videre til ***bcrypt.hash(req.body.password, salt, null, function (err, hash)***, hvor hashes til salt + password findes og gemmes i ***req.body.password***, for at næste middleware funktion kan bruge den. Resultatet kunne se således ud:



Algoritme, cost og salt gemmes sammen med hashen for at det skal være muligt at tjekke om en vilkårlig streng passer til hashen.

En streng skal tjekkes sker i når en bruger forsøger at logge ind, og her køres denne funktion (***comparePassword***), som får den streng der skal tjekkes samt algoritme, cost, salt og hashed password (ligger i ***hashword***)

Det der herefter sker er at bcrypt bruger algoritmen, cost og salt til at lave en hash til den streng der skal tjekkes. Hvis den matches med hashen fra databasen, er passwordet korrekt.



### MySql server

Til backend skal der bruges en database server der kan tage sig af lagring af alt bruger relateret data (som brugerne sender), samt holde styr på relationer mellem brugere og mellem brugere og aktiviteter.

Derudover skal database være punktet hvor foreslået matches bliver genereret. Dette skal gøres hver gang en bruger oprettes. Derudover skal databasen kunne kører op i en container for nemt at kunne kører hvor som helt samt skalere efter hvor meget trafik der er.

Derudover skal det være muligt at tilgå databasen via Express JS, da brugerne ellers ikke kan få deres information gemt, så det senere kan tilgås hvor som helst.

Ud fra de behov der er opsat er MySql valgt som den primære database i backend opsætningen. MySql er valgt fordi den kan løse alle behovene, er meget brugt og fordi der haves erfaring med den fra tidligere projekter. En anden kandidat var SQL server, men da disse 2 kan stort set det samme (ift hvad det skal bruges til i dette projekt), faldt valget på den det haves erfaring med.

I MySql skal der derfor designes en løsning, hvor der kan oprettes brugere. Brugere skal kunne have en relation(altså være venner) samt at der skal kunne findes foreslået matches for brugeren.

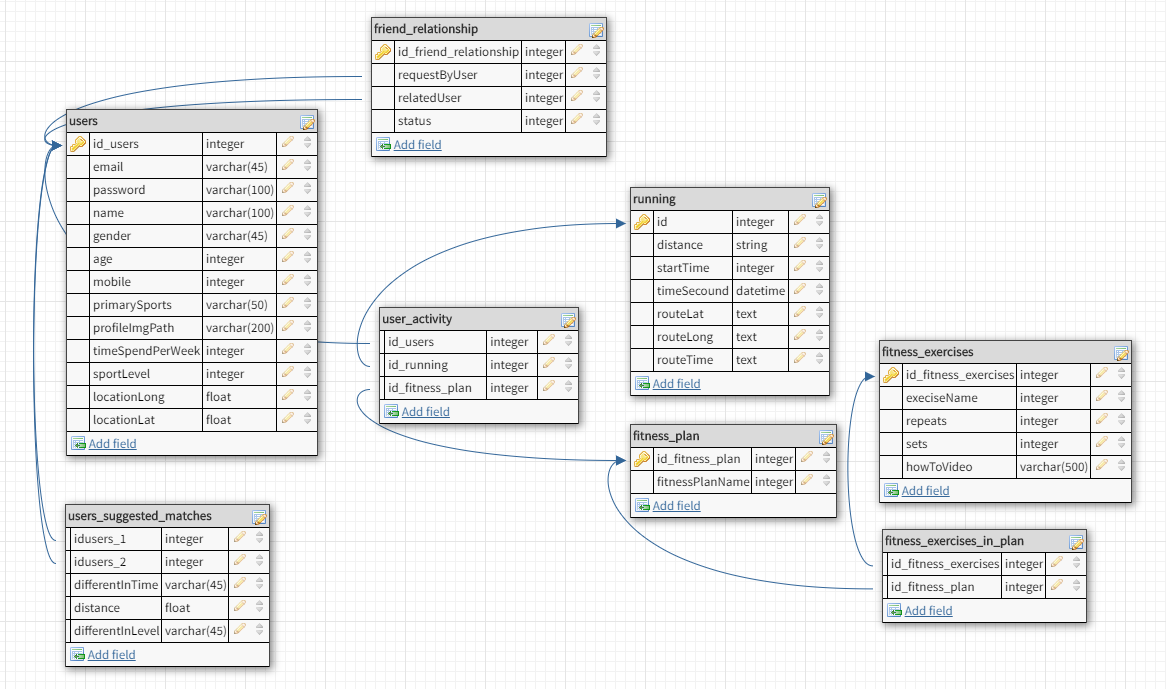
Derudover skal det i prove of concept løsningen være muligt at udfører et løb. Her skal de meste relevante informationer gemmes, altså starttid, længden af ruten, tid for gennemførelse og selve ruten koordinater.

Det skal også være muligt at oprette en fitness plan, men nogle forud indsatte øvelser.

Begge disse ting skal kunne referere til en bruger og man skal kunne dele dem med sine venner.

Ud fra kravene er følgende designløsning udarbejdede:

#### Design



Design løsningen gør brug af mange many-to-many relation-tables (users\_suggested\_matches, friends\_relationsship, user\_activity og fitness\_execises\_in\_plan), samt referencer (foreign keys) fra table til table.

Tabellen ***users*** indeholder alt information om en bruger. De ting brugere matches på er ***timeSpendPerWeek***, ***primarySports, sportsLevel og location***. ***profileImgPath*** indeholder en url til den specifikke bruges profilbillede (som er gemt på web serveren).

Tabellen ***users\_suggested\_matches*** fyldes med data alt efter hvordan brugeren passer til hinanden. Dette sker via en trigger som beskrives i næste afsnit.

Tabellen ***friend\_relationship*** indeholder data om relationer som venner i applikationen. En række oprettes når en bruger ansøger om venskab ved en anden bruger. Her sættes ***requestByUser*** til den bruger der ansøger, mens ***relatedUser*** sættes til der bruger der ansøges. Samtidig sættes ***status*** til 0, for at signalere at den at venskabet er i pending state. Hvis venskabet bliver godkendt af ***relatedUser*** bliver ***status*** sat til 1, ellers slettes rækken.

Tabellen ***user\_activity*** er for denne prove of concept designet således at id\_users kan referere til en bruger og running eller fitness plan. Her skal findes en mere skalerbar løsning i fremtidig udvikling, da der ellers skal være en kolonne for hver enkelt aktivitet, hvor kun 1 af dem er udfyldt i hver række.

Tabellen ***running*** designet sådan at et løb gemmes som en text i henholdsvis routeLat, routeLong samt routeTime. Her gemmes de som kommasepareret koordinater, som “pakkes ud” af frontend når de skal bruges til at vise ruten igen. running kan referere til en bruger igennem ***user\_activity.***

Tabellen ***fitness\_plan*** indeholder de fitness planer der er blevet oprettet i applikationen. Disse kan referere til en bruger igennem ***user\_activity.***

Tabellen ***fitness\_execises*** indeholder i denne prove of concept nogle forudindtastede øvelser som kan bruges i en fitness plan.

Tabellen ***fitness\_execises\_in\_plan*** indeholder information om hvilke øvelser der ligger i en fitness plan

#### Foreslå matches - Trigger

på MySql serveren kører der en trigger hver gang en bruger er blevet oprettet. Dette sker for at finde potentielle træningspartnere til brugeren. En brugers potentielle træningspartnere findes efter sport, tid brugt per uge, sports niveau samt placering.

I denne prove of concept løsning bliver brugerne potentielle matches hvis:

* De har samme sport.
* Hvis deres tid brugt per uge ikke er 40% mere eller mindre.
* Hvis deres sports niveau (1-10) ikke er 40% mere eller mindre.
* Hvis der ikke er mere end 20 km imellem brugerne i fugleflugt.

Alle 4 kriterier skal være opfyldt. Dette kunne gøres til variabler i fremtiden, således at brugerne selv kunne bestemme hvilke kriterier de vil matches på samt hvor store tolerancerne skal være.

Selve triggeren ser således ud:



Kort beskrevet findes matches blot ud fra kriterierne ovenfor, hvorefter de indsættes i ***users\_suggested\_matches.***

### Docker

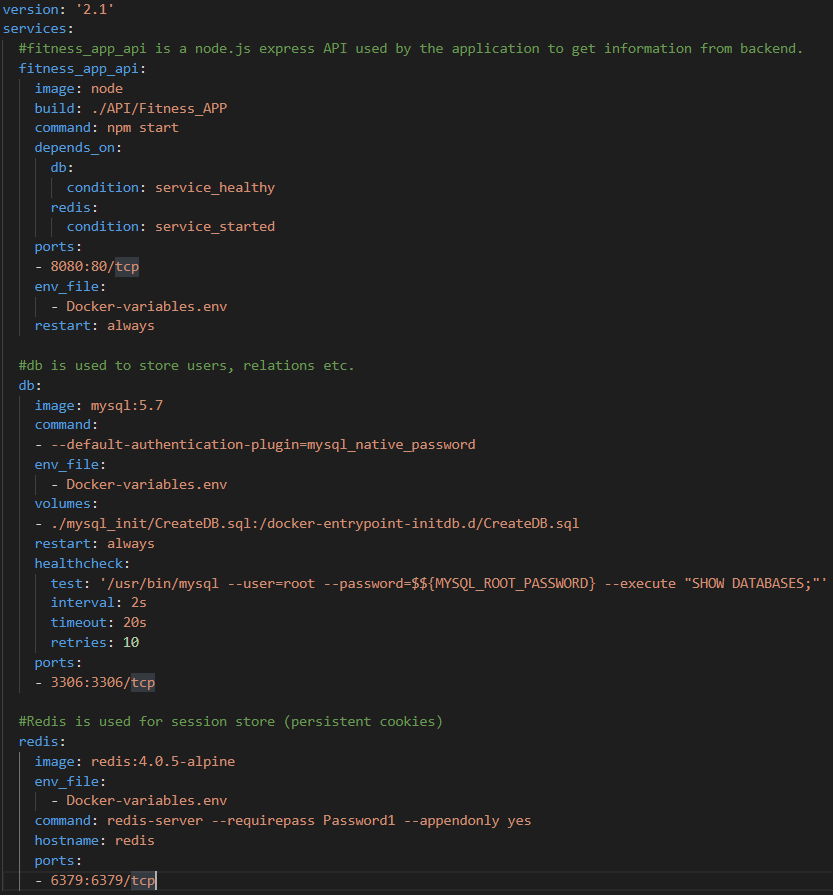
Docker er et værktøj, som gør det muligt at lave og kører en specifik applikation op på en bestemt måde. Dette sker i en container, som er isoleret fra det system den kører på, lidt ligesom en virtuel maskine. Forskellen på en virtuel maskine og en container ligger dog i at det ikke er nødvendigt at kører et helt operativsystem op, for at kunne kører applikationen. Her kan en del af den kernel som er på host maskinen bruges, og derfor er det kun nødvendigt at tilføje det der skal bruges for at kører applikationen op. Dette betyder meget store performance forbedringer ift. virtuelle maskiner.

Derudover er Docker designet til at kunne kører multiplatform, og derved kan samme miljø køres op på både Windows, Mac og Linux.

#### 

#### Docker compose

Docker compose bruges for at kører flere containere op på en gang. Dette bruges for at gøre deployment så brugervenligt som muligt.



Her vises docker-compose filen. I denne kører de 3 forskellige services op, nemlig API, MySql database og Redis. Opsætningen der sker kan forklares således:

API: Bygger en container ud fra node image samt source koden. Herefter ventes der på at MySql og Redis er oppe og kører (Da APIét afhænger af bruger databasen og sessioner). Når det hele er klar kører APIét op og kan tilgås fra port 8080 lokalt (port 80 forwardes til port 8080 via firewall).

MySql: Bygger en container ud fra MySql image. Derudover laves root brugeren og databasen køres op i den rigtige strukture. Docker tjekker om MySql er klar ved at tjekke på “SHOW DATABASES” kommandoen.

Redis: Bygger en container ud fra Redis image. Password sættes og derefter en session databasen klar til brug.

## Frontend:

### Swift:

Swift er apples eget programmeringssprog, som de har udviklet til IOS udvikling, vi har valgt at bruge swift til udviklingen af appen pga. følgende punkter.

Swift er hvad apple anbefaler.

Apple siger at Swift er 2.6 gange hurtigere end objective-c, selv om at nogle siger at det ikke er så meget. swift og obj-c bruger begge den samme IOS SDK og lov level virtual machine compiler. forskellen mellem sprogene er.

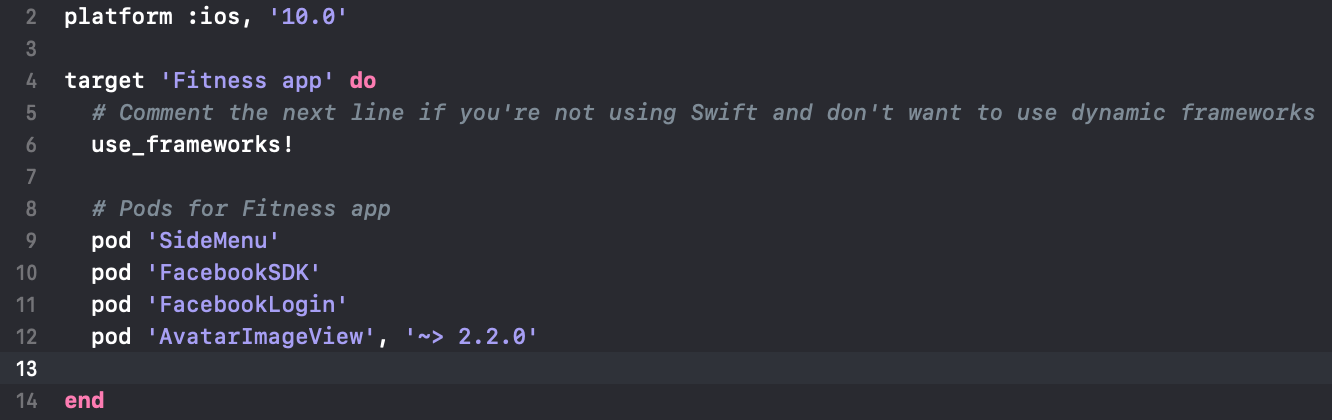
objective-c bruger en runtime code compilation, hvilket er at fx hvis obj-c object kalder et andet objekt så skal den køres et ekstra kompilerings sekvens før at koden er klar til brug. Det sker normalt rigtig hurtigt men når det gøres mange gang så tager det lidt tid.

Swift derimod er et godt valg inden for performance-sensitive code. Swift viser performance tal tæt på c++, selv om at swift er et forholdsvis nyt sprog, og derfor kan man også forvente at det bliver hurtigere med tiden.

Den største grund til at vi har valgt swift som sprog er sikkerheds delen af swift. Swift er blevet lavet til at forbedre sikkerhed i IOS apps og det er lavet som et type-safe og memory-safe programmeringssprog. Memory-safe hjælper med at undgå sårbarheder med uninitialized pointers. Disse typer for fejl er mest almindelige fejl og sværest at finde og debug. En stor grund er også at swift er meget nemt at skrive og forstå, fordi der ikke bliver brugt pointers, derfor er det nemt at finde fejl i appen. swift er også nemmere at servicere og holde ved lige idet at det er single program code file. hvilket kan ses i obj-c som værende 2 filer altså Objective C header (.h) file og implementation (.m) de 2 filer er lagt sammen i en fil (.swift).

### cocoapods:

cocoapods er en dependency manager for både swift og obj-c, det er bygget med Ruby, det cocoapods gør er at installere biblioteker til dit xcode projekt. det vi har brugt cocoapods til er at hente Facebook SDK som gør hele facebook integrationen mulig, og vi har brugt et biblotek der hedder sidemenu som hjælper med at lave vores sidemenu i appen. for at bruge cocoapods skal man først køre “sudo gem install cocoapods”, som installere selve cocoapods på maskinen. efter det skal man lave en podfile i projektet som kan se sådan ud.



Det der sker i denne podfile er først definere du hvilket laveste ios der kan bruges i den appen, efter det vælger man et target, så skal man have dynamic frameworks i swift. dynamic framework er lavet til at reducere loading tiden for en app of den hukommelse som en app skal bruge, også er selv appens executable fil blevet gjort mindre end i statisk framework.

efter use\_frameworks! kommandoen skal man skrive de pods som der bruges i appen, på bildet er podfilen for fitness app'en, derfor er SideMenu, facebook og AvatarImageView som vi ser nærmere på senere. og så sluttes filen af med “end” der efter kørers der en pod install i en comandline. også bliver der lavet en .xcworkspace af dit xcode projekt og en pods mappe.

### Facebook sdk:

vi har valgt facebook login for det er et stort socialt medie, som har ufatteligt mange brugere, de har en stor mængde data om der brugere som vi kan bruge i appen, ud over det er der mulighed for at folk der bliver træningspartnere, kan finde hinanden over facebook. Ud over det kan man med facebook sdk implementere messenger i app’en sådan folk kan kontakte hinanden over facebook hvis der er interesse for at udføre en træningsform sammen.

Der er også andre alternativer såsom google login som bla. bruger gmail som login, typisk er der bare ikke så meget data at få fra den kant da folk typisk ikke giver den konto alt for meget data om dem selv.

Facebook sdk er en facebook integration som facebook laver til IOS og mange andre platforme, i facebooks sdk for swift kan man bla. lave login, dele og sende beskeder, log ting fra appen, så som hvor mange brugere der bruger app'en med facebook login, og en masse andre parametre kan der kigge på. der ud over kan man bruge graph API som bruges til at query data, udgive nye stories, uploade billeder og en masse andre ting. Det er bla. graph api som bruges til at få de informationer omkring brugeren som vi skal bruge for at kunne oprette brugen i databasen.

implementeringen af facebook

først skal disse 3 pods addeds til podfilen



så skal man køre en pod install, efter det skal man lave en app inde på facebooks developer site, når det er gjort kan man følge quickstart under facebook login under products under ens app. der skal man bla. connect dit xcode projekt og din app hos facebook med dit bundle id som er et unikt id som bliver sat sammen af firmanavnet og app navnet. derefter skal man vælge noget settings hos facebook. derefter begynder man selve opsætningen. først opsætningen af plist hvor man skal have facebook app id’et og facebook navnet som skal vises når man spørger brugeren hvilke informationer men gerne vil have. derefter sættes appdelegate klassen op. men dette step er lidt anderledes idet at facebook på nuværende kun har lavet en obj-c appdelegate, så den skal skrives om til swift. dette kan gøres sådan.

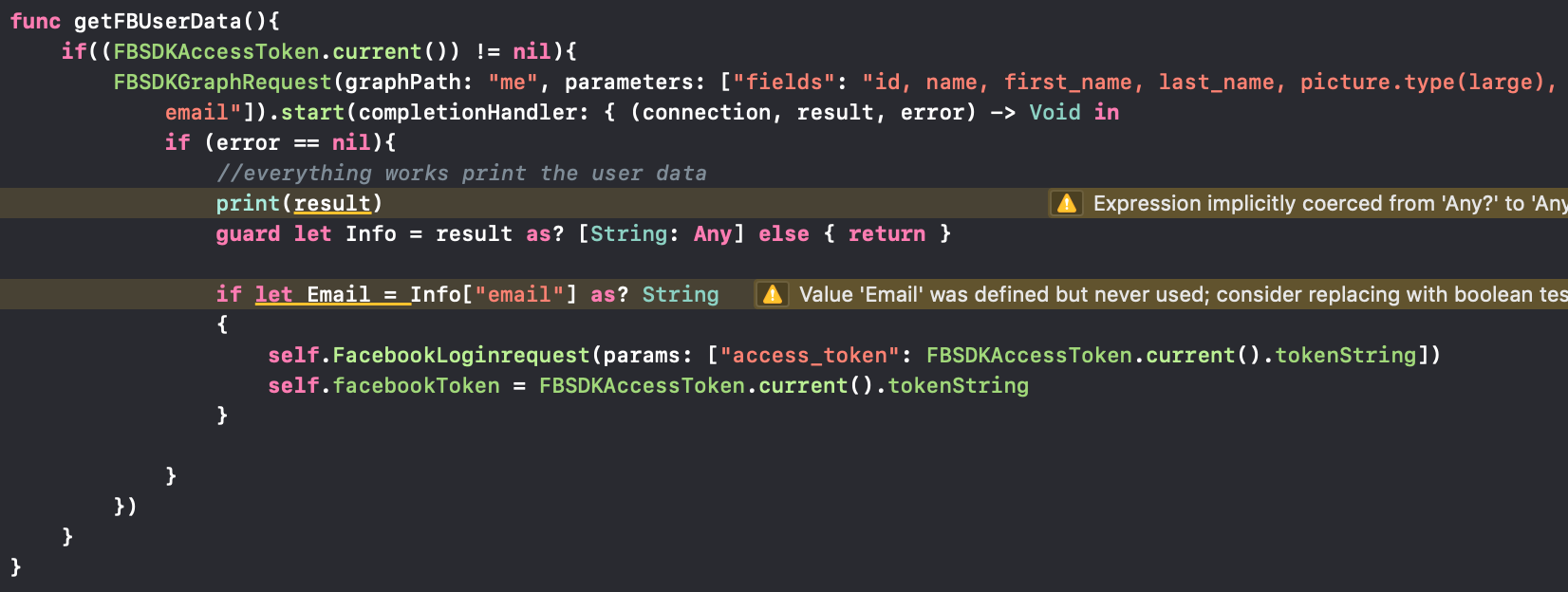


Det der sker her er at her er når app'en åbnes og er færdig med at loade så køres så loades facebook ind i appen. efter dette kan selve ui integration påbegyndes.

For at have flere frie tøjler i forhold til layout så har vi valgt at lave en custom login button. derfor laves der en UIbutton action som ser sådan ud.



i toppen af .swift filen laves in import FBSDKCoreKit og import FBSDKLoginKit hvilket giver mulighed for at lave en FBSDKLoginManger som er den manager der skal til at lave et login forsøg med facebook, derefter laver man et login forsøg med den email som brugeren har indtastet i det facebook webview som vises for brugeren, der efter får man et resultat som enten er granted Permissions eller også har brugeren afbrudt eller sagt nej det forespørgslen for hans profil data. hvis brugeren har sagt ja, så køres en facebook funktion som er den funktion som henter den data som brugeren har sagt ja til og dermed også den accesstoken som skal sendes til api’et for at vi kan få den session med vores api og database.

i vores getFBUserData gør vi som følger. 

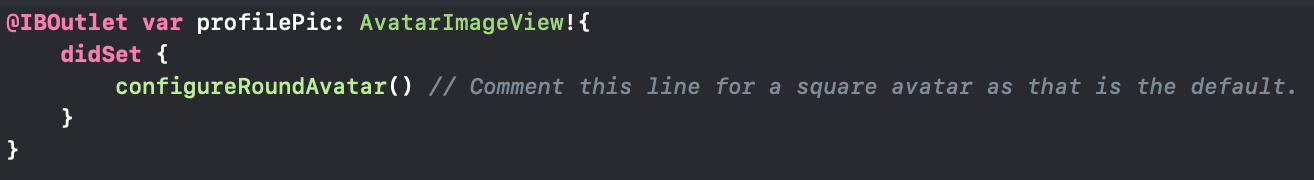
grunden til at vi spørger om den data i appen er for at få den rigtige facebook access token så vi kan få den rigtige da til vores backend, som henter dataen efter brugeroprettelsen. Vi kan ikke få alle de data vi gerne vil have fra facebook, da facebook af gode grunde ikke spørger om hvilken type fitness en bruger går til og hvor gode de er til det. derfor sender vi brugeren videre til et step 2 i oprettelsen for at få de sidste data. og det gør vi sammen med facebook accesstoken, som vi sender med til vores andre info til api’et.

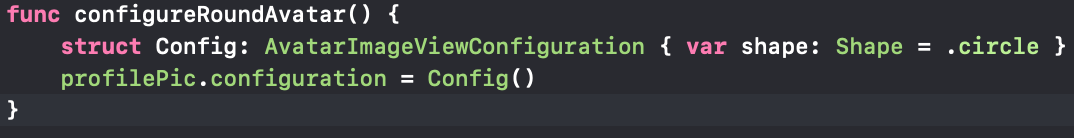
### AvatarImageView:

AvatarImageView bruges udelukkende til UI, det er et bibliotek der muliggør det at lave runde billeder i xcode, vi bruger det derfor kun til at lave en flot profilbillede. Grunden til at vi har valgt at bruge AvatarImageView og ikke selv bygge et bibliotek fra bunden er, det er en stor opgave at lave det men der er ingen grund til at lave det selv når AvatarImageView giver alle de muligheder som vi gerne vil have og så er alle de bugs der er fra starten de er fixed, dem skulle vi ellers selv til at fixe og bruge tid på.

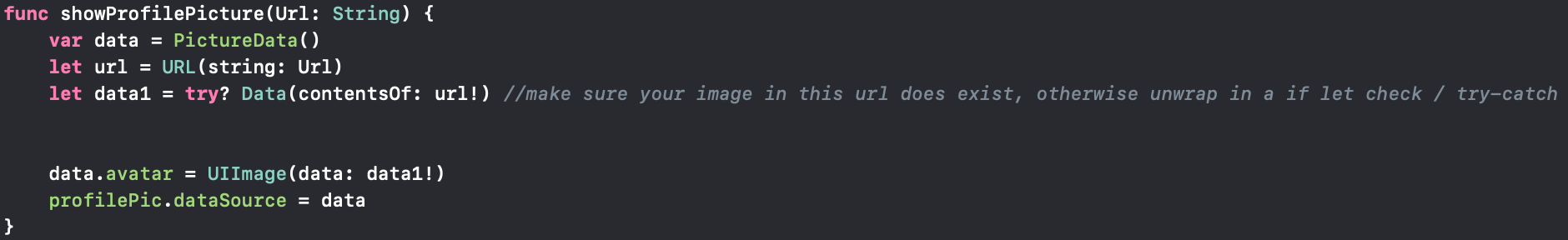
Der er 2 forskellige implementeringer af dette bibliotek, en når det bruges i en view controller og en anden når det er i et tableview.

De to implementationer ser sådan ud.

Først sættes uiImageview class til AvatarImageView og derefter laves der en outlet af det imageview. 

også sætter man en setting på det imageView, i vores tilfælde vil vi gerne have en rundt billede derfor laves der en funktion med de settings der gør det rundt. 

først laves en struct som holder den konfiguration i den struct laver vi en variable som holder .circle som en type Shape. Typen Shape er en type som bliver defineret i AvatarImageViews pod som vi har includeret i projeketet. og der efter sætter man konfigurationen til uiimage.



når et profilbillede skal vises skal vi have dataen til profilbilledet, og det får vi i form af en url til en sti på en storage server, den url skal laves om til data(billede) sådan det kan vises i vores imageview. men først skal man lige have den data lavet om til en AvatarImageViewDataSource, og det gøres i en struct som vi kalder data og der sætter man avatar til at være billeddata, også kan man til sidst sætte billedet ind i uiimageviewet.

### URLSession:

vi bruger URLSession til at hente og sende data til vores backend, URLSession er en class der giver et api der kan hente send data til endpoints på urls, og det kan man både gøre i baggrunden når appen ikke køre, fx. hvis vi vil lave popup beskeder omkring at en anden bruger vil være venner med dig.

Der er flere forskellige configurations typer for en URLSession.

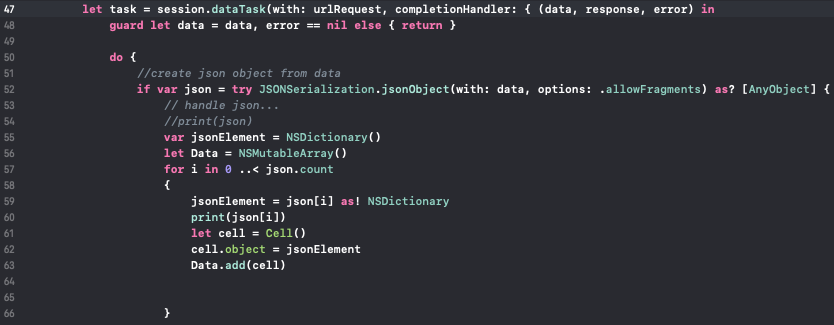
Der er en shared session som er en basic type session den tager ikke imod configuration, hvilket betyder at man ikke kan customise det. vi bruger den type session nogle steder i appen.

En default session som vi også bruger iappen, ligner meget en shared session men der er mulig for at lave mere configuration.

Der er også flere typer for URLSession tasks, vi bruger dog kun en type som er Data task som er en session type hvor det både er muligt at send og modtage data, hvor man får dataen i et NSData object, denne type session task bliver brugt når det er små og kald der bliver kørt ofte. i alle typer kald der bliver lavet sendes noget data med også fås der et respons i form af data der skal bruges i UI.

Responsen bliver enten kørt gennem en class eller lavet om til et json object.

Når data’en bliver lavet om et json object sker det sådan her.

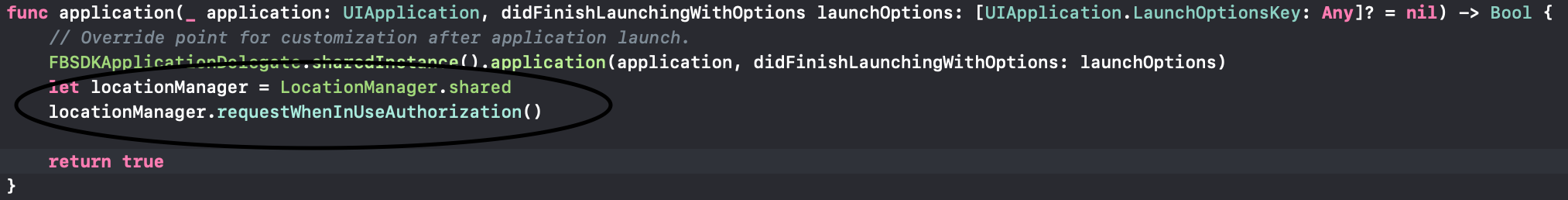


først tjekkes om der noget data, det gøres med en gaurd som returner og stopper funktionen, der efter prøves der at lave en JSON Serialization som laver data om til et jsonObject og det er så af typen AnyObject, grunden til at det er af typen AnyObject er at så er man ligeglad med hvad der kommer, det er noget man kan gøre hvis man ved hvad der kommer fra backend. Efter at hele dataen er lavet om til et json object, så loopers det gennem alle objekter for at lave et objekt til hver cell i tableviewet.

URLSession understøtter data,file, ftp, http og https URL schemes som standard. vi bruger http i vores koncept app, fordi vi ikke har noget ssl certifikat.

### Location tracking:

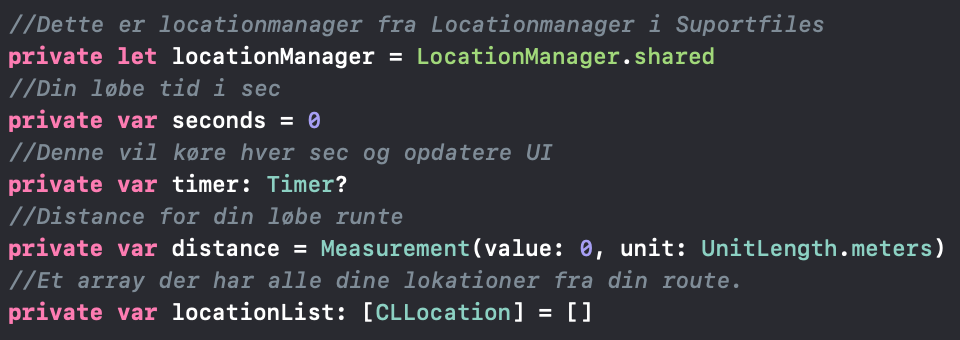
I appen har vi en running section, hvor en bruger kan track en løberute til det skal vi bruge en LocationManager, som kan få mobilens lokation, men før at man kan få en mobils lokation skal man spørge brugeren om lov til det. dette gøres på denne måde i appdelegate



her spørges der om man må bruge mobilens lokation når appen er i brug.

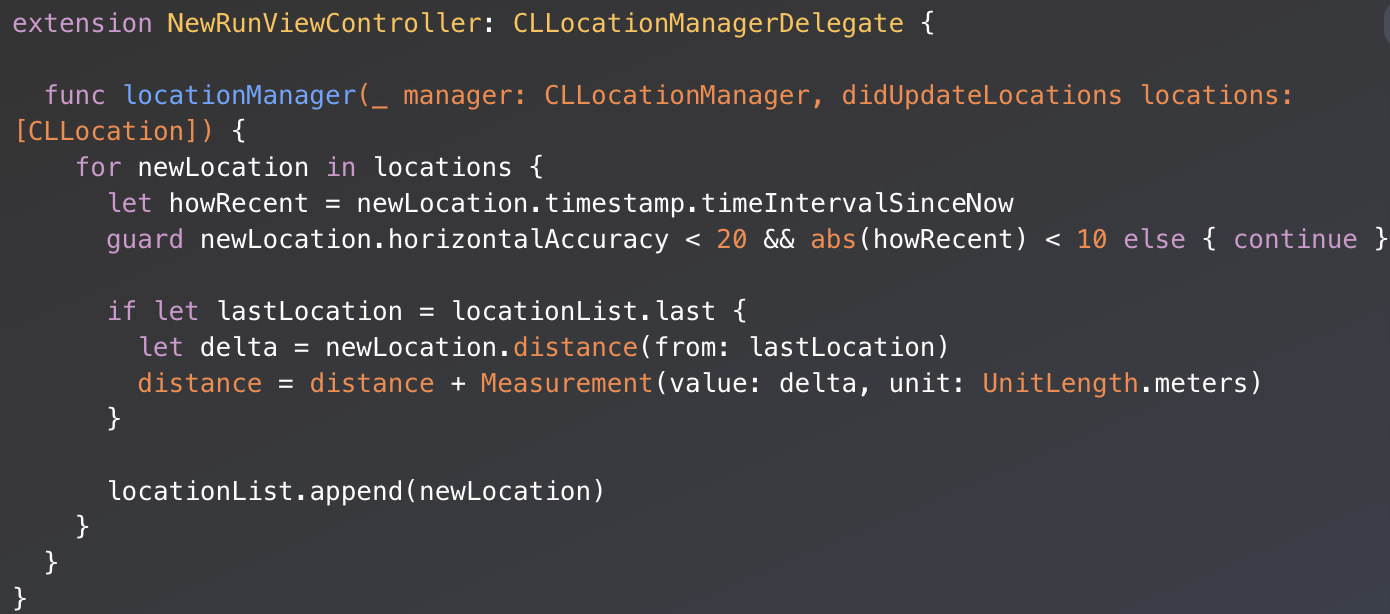
hvis brugeren går ind i running sektionen så kan brugeren få tracket sin route hvis brugeren har sagt ja til requesten fra locationManageren.

#### opsætning af tracking af en route:

først starter man med at en import CoreLocation, efter det laves alle variabler som skal gemme alle infomationer.  


codekommentarene forklare variablerne.

så laves en extention til vores viewcontroller som CLLocationManagerDelegate. som ser sådan ud.



grunden til denne extention er, Core Location rapporterer positions opdateringer via CLLocationManagerDelegate. Denne Delegate metode bliver kaldt hver gang Core Location opdaterer brugerens placering og laver et location object. dette array som der bliver tilføjet til i slutningen af Delegaten indeholder flere lokationer, som bliver listet efter et timestamp på lokationen.

En CLLocation indeholder nogle gode oplysninger, herunder længdegrad, breddegrad og tidsstempel for positionen. i linjen

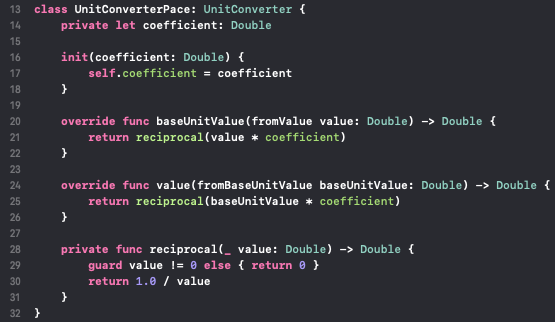


bliver lokationen tjekket for nøjagtighed hvis telefonen ikke er sikker på at lokation ikke er inden for 20 meter for den nøjagtige position så bliver den ikke tilføjet til listen med lokationer. ud fra disse informationer kan distancen og hastigheden beregnes. dette gøres i med UnitConverter, som er næste afsnit.

### UnitConverter extention(apple) :

unitConverter extension er en extention der har formålet at konvertere de data der opsamles af vores location tracking, sådan at brugen kan få hvilken hastighed de løber med omregnet til pace regnet i end min/km eller min/mile, alt afhængig af hvor brugeren befinder sig henne.

For at gøre dette skal man have matematikken på plads for hvordan man regner dette ud. den måde som vi har valgt at gøre det på er med en extention af apples allerede meget dækkende unitConverter, i det at det er den nemmeste og bedste måde at gøre det på idet at det er bygget ind i alle de komponenter som vi bruger for at få brugeres lokation under et løb.



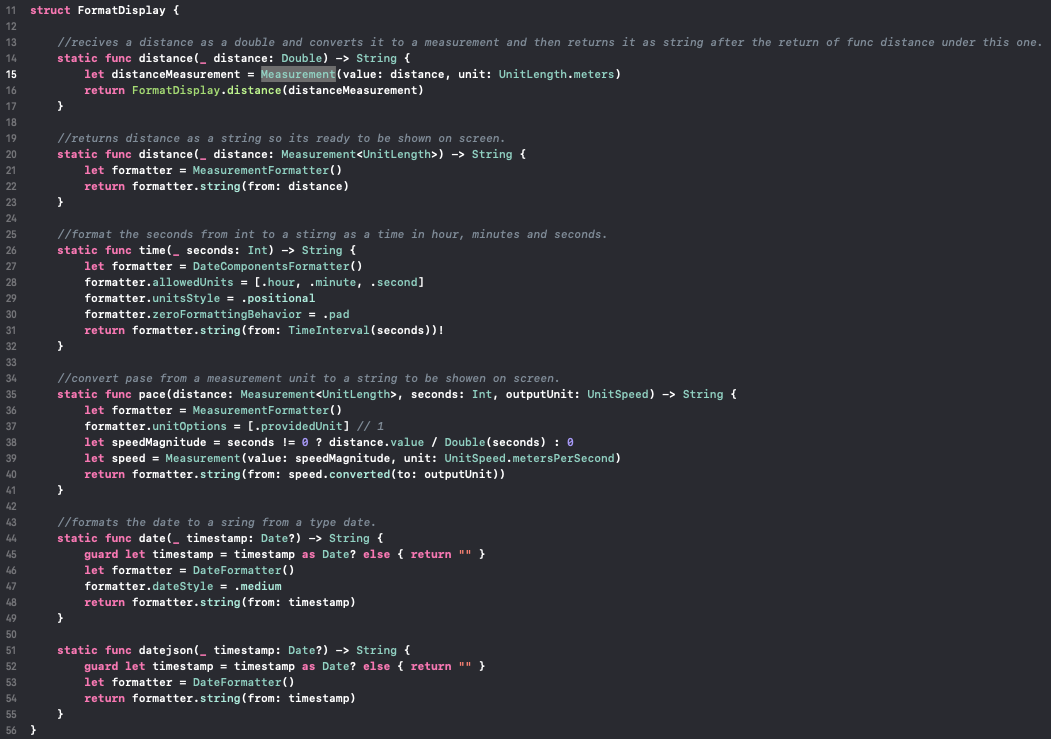
Denne unitConverter tager sig af matematikken med at omregne vores Unit speeds. hvilket vil sige at den regner vores hastighed ud i den valgte unitspeed, som bliver defineret under unitConverterPace klassen.



Her laves selve extentions af UnitSpeed, UnitSpeed er en af de mange typer enheder, der leveres i Foundation. UnitSpeeds standard enhed er "meter/sekund". men denne udvidelse gør det muligt at udtrykke hastigheden i min/km eller eller andre typer. coefficienten bestemmer hvilken type det, første del af coefficienten er tiden altså de 60.0 og den sidste er antallet af meter hvilket er de 1000, men hvis det var miles man skulle regne om til ville det være 1609.34 da det er det antal meter der er på en mile.

### FormatDisplay:

Du har brug for en ensartet måde at vise mængder som distance, tid, tempo og dato i App’en. MeasurementFormatter og DateFormatter er inbygget function fra apple der kan hjælpe med at vise data’en i den rigtige format. Dette vil gøre UI hurtigere og mere stabilt. Dette kan opnås med nogle funktioner som klare formateringen.



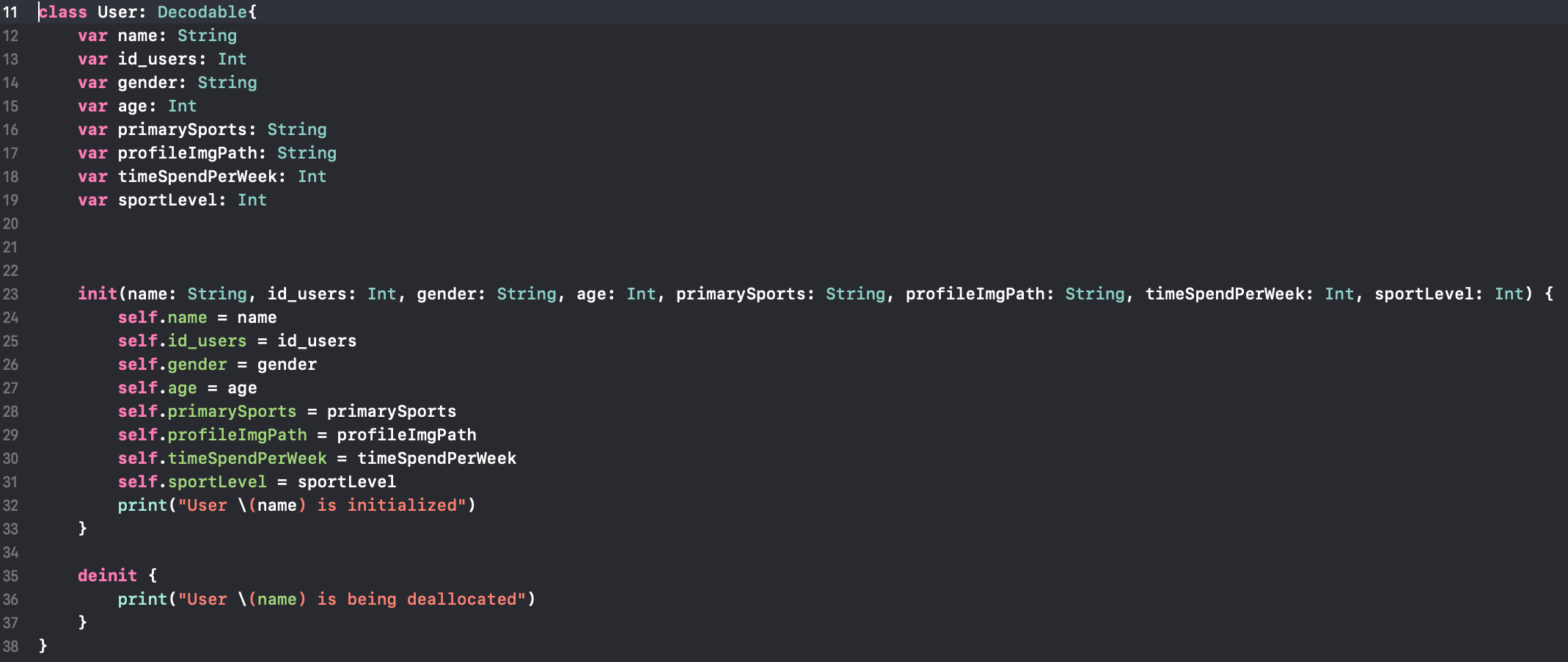
i denne struct er der flere funktioner som skal konvertere data til at blive klar til at blive vist på en viewcontroller eller tableviewcontroller, den første funktion tager distance der er mål i et løb bliver konverteret til en measurement med meter som målestok. Derefter bliver den lavet om til en string som er nemt at vise. det gør den med en measurementFormatter som er en apple class indbygget i ios 12.

Derefter går har vi en time formater som ændre en int der er de antal sec som et løb har taget til en string der har en format af timer:min:sec. Det gøres med en DateComponentsFormatter som man kan sætte date formatet på, i dette tilfælde vil vi gerne have timer:min:sec idet at et løb aldrig tager mere end 24 timer. et eksempel på hvordan formatet vil se ud kunne være, en løbetur på 5640 sec så vil denne funktion omregne det til 1:34:00 hvilket er 1 time og 34 min.

når pace skal gøres klar til til at vises på viewcontrolleren, så køres pace funktionen den modtager distancen, løbetiden i sec og hvilken unit men gerne vil have pace i, også returnere den en string, som er pace i min per km. for at få hastigheden ud skal man først lave en MeasurementFormatter også derefter skal speed udregnes efter distances der / med det antal sec der er gået så har man antal meter pr sek og derefter bliver speed så lavet ud fra hastigheden, også formatere man det hele til en string. som kan vises i UI.

så er det at konvertere en dato til en string, det gøres på følgende måde, først må man være sikker på at det er en dato det man har, efter det laves en DateFormatter også der sættes hvilken længde på datoen der ønskes. en medium er fx. Nov 23, 1937. også returneres en string bestående af datoen med en medium længe.

### class User: Decodable:

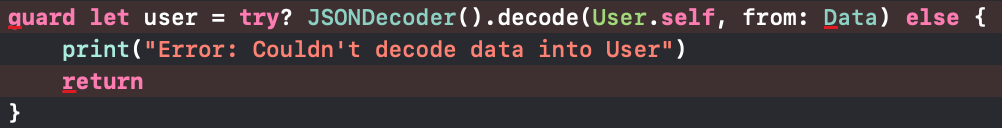


Dette er vores class user, som bruges til at holde på users til vores tableviewcontroller som skal vise brugere, dette tableviewcontroller kan fx. være vores find mulige træningspartnere. Bemærk, at klassen er med Decodable-protokollen, så vores JSON element som vi får fra backenden kan konverteres til denne klasse-instans.

lad os sige at vi har lavet en bruger med følgende info og vi henter brugeren fra backenden.



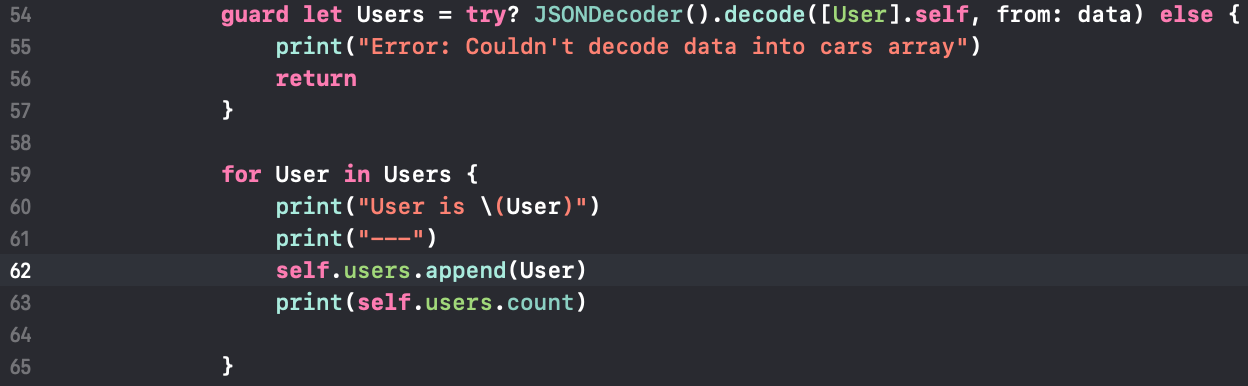
så vil vi lave en JSONDecoder



JSONDecoder vil decode JSON dataene i User klassen, User.self fortæller JSONDecoder at denne JSON indeholder egenskaber fra User klassen og afkode den i User klassen.

Der er en try? foran JSONDecoder, da de data, der sendes til dekoderen, muligvis er en ugyldig JSON eller slet ikke en JSON. Når det sker, vil JSONDecoder().decode metoden smide en fejl. Try? bruges til at fange den fejl, hvis fejlen er blevet kastet.

Da dette ved et enkelt json element er det rimeligt nemt, men når vi laver vores tableviews så vil vi gerne have flere brugere sådan man kan se alle de brugere som man muligvis matcher med, så ville der blive sendt en array af json elementer, så bliver vi nødt til at loope igennem alle elementer, også skal alle elementer putte i et array af det User classes. og det gøres på denne måde.



[User] betyder at det er et array af class User, [User].self fortæller JSONDecoder at det JSON Data har et array af User class. hvis User er Decodable, så kan [User] også Decodable og kan derfor også blive decoded. og så kan vi loop gennem arrayed og append det til et array af brugere som vi kan putte i vores tableview.

### Profilbillede base64:

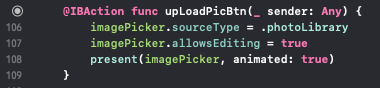
når en bruger vælger et profilbillede fra deres telefon, starter man med at lave en imagePickerController, som er et apple view af dit fotobibliotek. Dette er den eneste måde at få adgang til en brugeres fotobiblotek. Dette gøre på denne måde.

først starter man med at definere UiImagePickerController

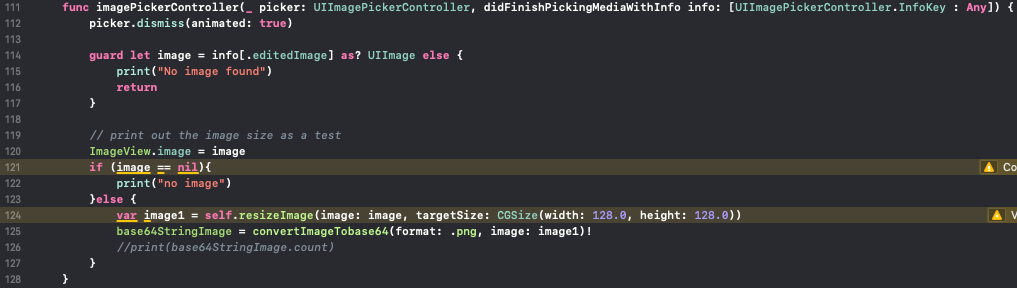
**let** imagePicker = UIImagePickerController()

Derefter åbner man imagepicker med sourceType, vi har valgt kun at bruge photo library men kamera kan også bruges, sådan at en bruger kan tage et nyt billede med deres telefon.

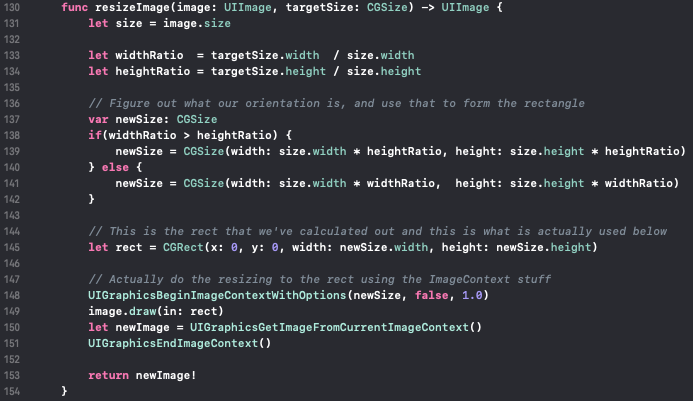
så kan man sige om man vil have at brugere kan redigere det billede brugeren har valgt.



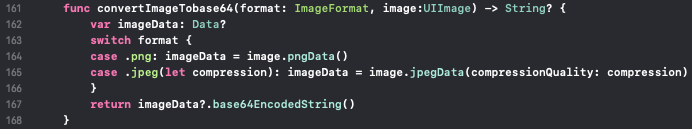
også sådan gøres det.



Denne funktion køres når brugeren har valgt et billede, først lukkes imagePicker viewet, så tjekkes der om det er et billede der er valgt, for at billede kan blive sendt i en post request, så skal billedet gøres mindre, sådan det ikke fylder så meget. dette gøres med en resizeImage Funktion som se her under.



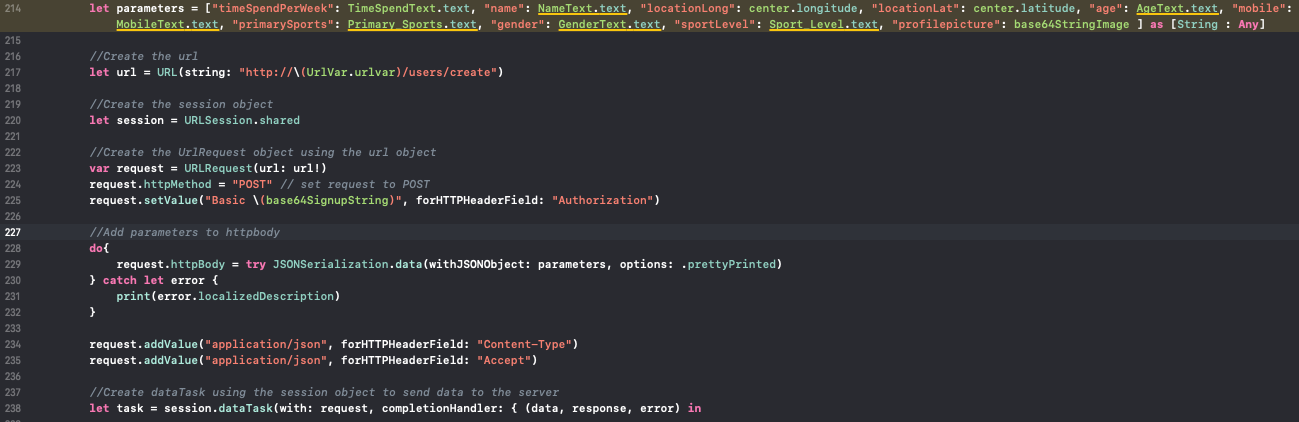
efter denne funktion er kørt har man så en resizedimage som kan konvateres med følgene funktion.



### Requestes:

i vores app bruger vi forskellige typer for requests, vi har post,put, get og delete. vi bruger disse forskellige typer for requests til bla. oprettelse af bruger, login, mulige træningspartnere og til svar af en friend request.

opsætningen af en request kan ses på nedenstående billede.

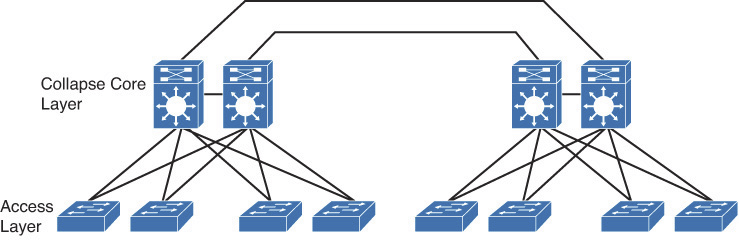


# Opstart af Fitness Junkies netværk

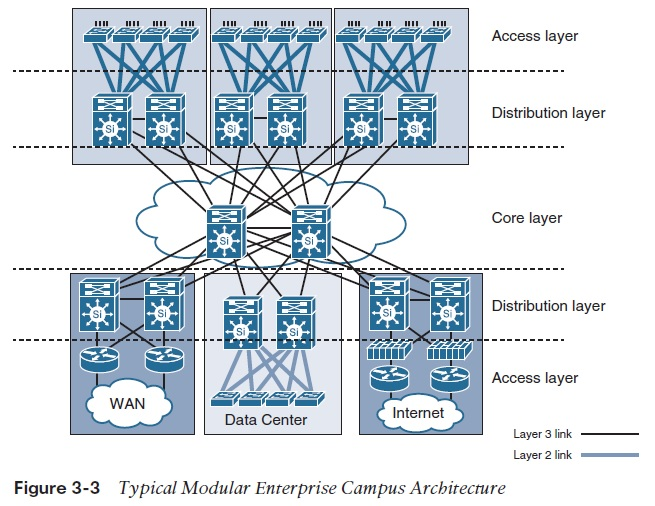
Vi vil gerne bygges vores bedste bud på et fuld funktionelt netværk med design, tilgængelighed, sikkerhed og skalerbarhed i fokus, så appen altid er kørende. Vi vil gerne designe et netværk efter Ciscos best practice, såfremt netværket bliver så godt som muligt.

## Design:

I Fitness Junkies vil vi lave et collapsed core design, da det passer bedst til virksomhedens størrelse. I et collapsed/core design slår man core og distributions laget sammen til et, hvilket gør at man sparer en masse penge for det første og da det ikke er nødvendigt med et helt core lag når man ikke skal connecte til andre branches. I et collapsed core design vil der også være mulighed for at skalere netværket op når der kommer flere ansatte.



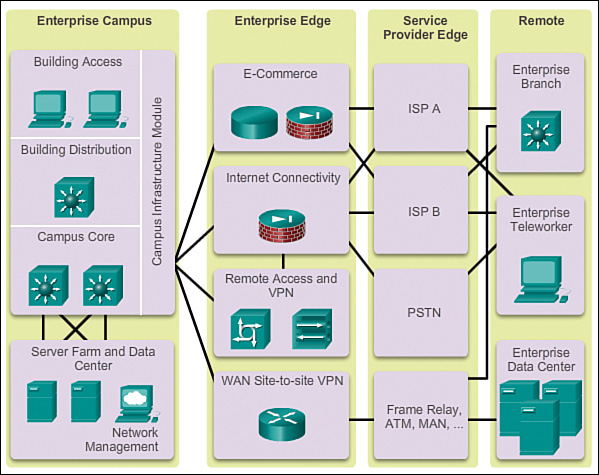
I større virksomheder ville man benytte sig af et three-tier design, da man er oppe i rigtig høje hastigheder, og at det er nemmere at connecte sites imellem hinanden.



På billedet ser vi et eksempel på et Enterprise Campus også kaldt three-tier design. Dette ville man typisk bruge i større virksomheder for som man kan se på billedet så bliver der connected flere switch blocke fra forskellige sites til core laget, hvilket er smart, fordi det gør kablingen nemmere. Hvis vi ikke havde et core lag ville vi skulle connect alle vores switch blocke på kryds og tværs, hvilket ville være uoverskueligt. Da vi kun har 1 site i fitness junkies benytter vi os a two tier modellen/collapsed core.

Ciscos netværks design er delt op i 4 moduler:

1. Enterprise campus som beskrevet overstående.
2. Enterprise Edge
3. Service providers Edge
4. Remote



På billedet her ser vi de 4 moduler. I fitness junkies benytter holder vi os til Ciscos design, men vi bruger ikke alle moduler som vist på billedet. Vi benytter os af:

Building Access:

Access laget er her hvor vi connecter end devices. Det er her brugerens computer kobles til en switch eller access point så han/hun får adgang til netværket. På access laget er protokoller som spanning-tree, storm control og port security vigtigt. Da man ikke ved om en bruger laver loops eller prøver at ødelægge netværket.

Building Core/Distribution:

I distribution/core laget holder vi blandt andet styr på vores routing og vlan. Her bliver der lavet routing så det er muligt for brugeren at komme lige hen til den destination han skal. I distribution/core laget sker alt det “hårde” arbejde, da det er her der foregår mest trafik og flest routing protokoller.

Server farm and Data center:

I server farmen hoster vi alle interne servere. Serverne er tilkoblet core/distributions laget.

Enterprise edge:

Enterprise edge er kanten på netværket. I Fitness Junkies har vi sat en Asa firewall op, som vi bruger til at lukke alt farlig trafik ude og lukke det trafik ind vi skal bruge til blandt andet appen. Derudover er der også lavet en DMZ zone som gør vi har en zone der er adskilt fra Lan.

Service providers edge:

Service providers edge er dem som levere internet forbindelsen også kaldt ISP. I vores proof of concept er det Mercantec.

## Valg af udstyr:

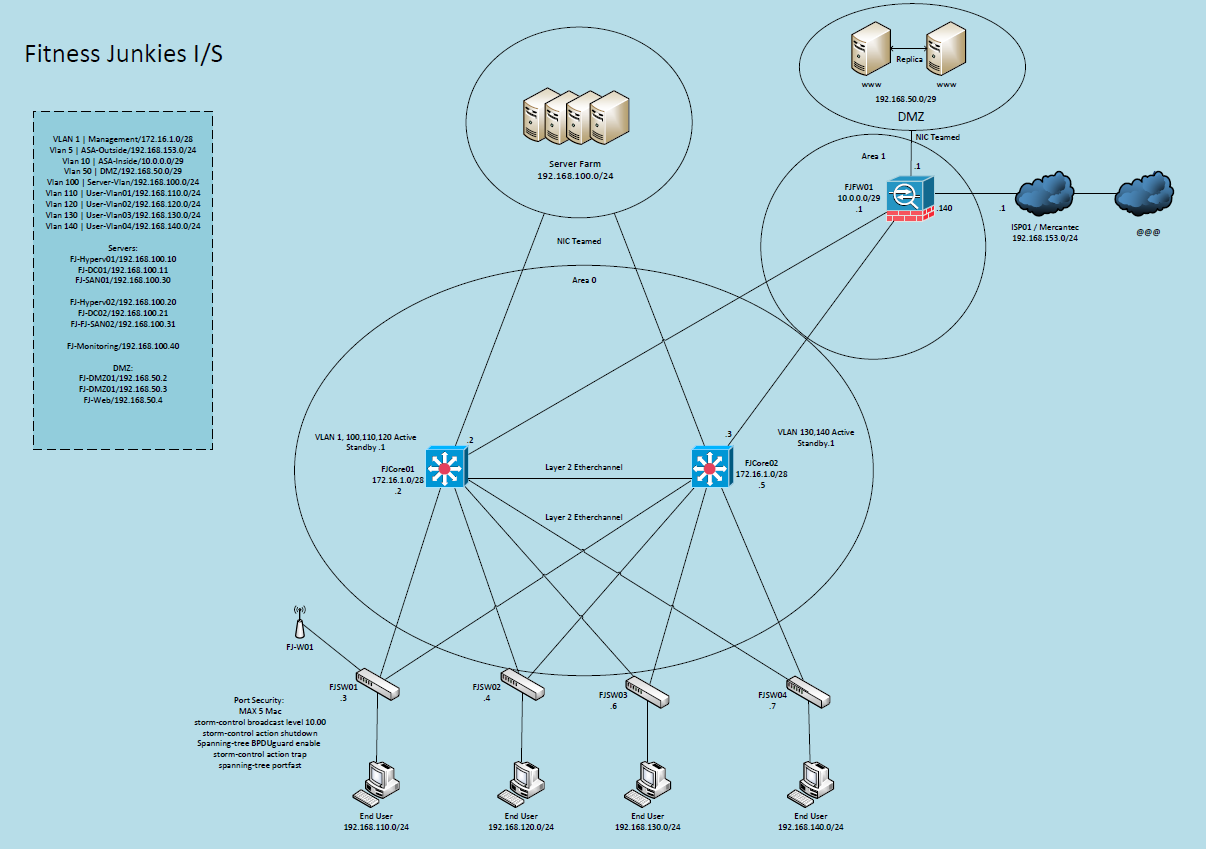
På access laget har vi valgt at bruge layer 2 switche(Cisco Catalyst 2960X). Layer 2 switche har ikke en indbygget router i sig, men de kan bruges til det formål at få end users på netværket. Man kunne også bruge layer 3 switche på access laget, men dette vil være en meget dyrere løsning til at løse præcis det samme formål.

Derudover er der også en access point(Cisco Linksys EA4500) så medarbejderne kan komme på det trådløse netværk. Der vil efterfølgende blive tilføjet flere access points.

På core/distributions laget har vi valgt at bruge layer 3 switch Til dette formål benytter vi os af Cisco Catalyst 3750. Man kunne også bruge en router. Fordelen ved at bruge en layer 3 switch er at vi kan route meget hurtigere da en layer 3 switch benytter sig af hardware til at route og switche trafikken, hvorimod en router bruger software. Vi har også mulighed for at lave etherchannel links til load balance, da vi har mange flere porte.

På Internet Edge bruger vi ASA5505 til at håndtere det trafik der kommer ind og ud af netværket i fitness junkies, samt lave vores DMZ zone. Ciscos ASA firewall er nemlig rigtig smart til at håndtere trafik og sørge for netværket er sikkert, da den kun tillader det trafik vi siger den skal tillade. Derudover har den også en funktion til at lave en DMZ zone som er en isoleret zone der isolere LAN segmentet fra det som er i zonen. Dette vil jeg komme mere ind på senere.

## Topologi(Topologi vedhæftet i bilag, så man kan zoom):



Topologi Forbedringer grunde ressourcemangel:

Da ASA5505 ikke kan lave HSRP, men kun et “failover link” som jeg ikke kan få til at fungere optimalt(Demonstrere det senere i rapporten) har jeg valgt at bruge 1 ASA i proof of concept. Havde man en ASA5506x kunne man lave HSRP med failover + Dual isp, så vi havde fuld redundans på firewalls. Derudover undersøtter ASA5505 heller ikke etherchannel. Derfor har vi en single point of failure i proof of concept løsningen på FW01. Jeg vil dog demonstrere hvordan man konfigurere det, da vi har brugt teknologierne på distribution/core laget.

## 

## IP Plan:

Vlan 1

Mangagement /172.16.1.0/28

FJ-Core01 .2

FJ-SW01 .3

FJ-SW02 .4

FJ-Core02 .5

FJ-SW03 .6

FJ-SW04 .7

Vlan 5/Outside /192.168.153.0/24

FJ-FW01 .140

Vlan 10/Inside /10.0.0.0/29

FJ-FW01 .1  
FJ-Core01 .2  
FJ-Core02 .3

Vlan 50/DMZ /192.168.50.0/29

FJ-DMZ01 .1

FJ-DMZ02 .2

FJ-Web .3

Vlan 100/Server-Vlan /192.168.100.0/24

FJ-Hyperv01 .10

FJ-DC01 .11

FJ-SAN01 .30

FJ-Hyperv02 .20

FJ-DC02 .21

FJ-FJ-SAN02 .31

FJ-Cluster .15

FJ-Hyper-v Broker .16

FJ-DFS Manager .17

FJ-Monitoring .40

Vlan 110/User-Vlan02 /192.168.110.0/24

FJSW01 Access DHCP.100-200

FJ-W01 .10

Vlan 120/User-Vlan03 /192.168.120.0/24

FJSW02 Access DHCP.100-200

Vlan 130/User-Vlan04 /192.168.130.0/24

FJSW03 Access DHCP.100-200

Vlan 140/User-Vlan01 /192.168.140.0/24

FJSW04 Access DHCP.100-200

## Opsætning af netværk

Til at starte med blev alt udstyret sat op fysisk og resat til factory, da udstyret i proof of concept løsningen har været brugt før. Dette gør vi ved at:

1. Hold boot knap 10 sek efter boot indtil den lyser amber og grøn igen
2. Gå ind i flashen med kommandoen: Flash\_init
3. Åben flash biblioteket: Dir Flash:
4. Slet alt config som de tidligere folk har brugt på den: Del flash:Config
5. Reset

## Subnets:

Som man nok har bemærket i IP planen har vi brugt forskellige størrelser subnets. Dette gør vi for at gøre vores broadcast domain mindre i det pågældende netværk. Derudover skaber det også sikkerhed da vi ikke har brug for flere enheder på de pågældende subnets. Er der brug for mere netværksudstyr i fremtiden bliver subnet konfigurationen ændret.

Tager vi fx 10.0.0.0/29 som er vores links imellem de to core/distribution switche og vores firewall.

Her har vi brug for 3 adresser. Det tætteste vi kommer på det er i et /29 subnet, hvor der er 6 ledige adresser. 10.0.0.0/29 består af et klasse a netværk. Vi kan opdele netværket i følgende dele

10.0.0(Netværk) | .0(Host)

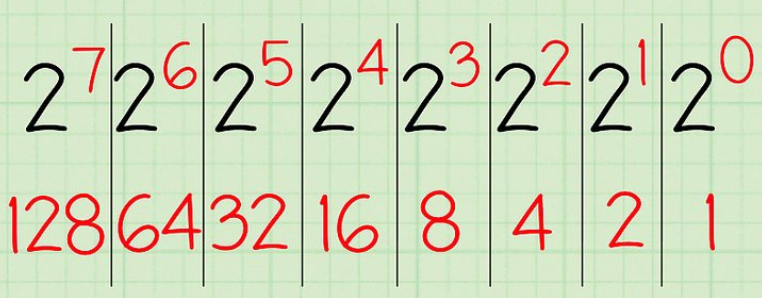
Subnet mask: 255.255.255.248 = /29

I binær :11111111.11111111.11111111.11111|000

De sidste tre 0 definere host og de resterende definere netværket

host i binær:

2\*2\*2 = 8



Da den første og sidste adresse er reserveret til henholdsvis netværk og broadcast bliver det til 6 tilgængelige adresser.

## Netværks konfiguration:

Vi starter med at konfigurere vores netværks udstyr. Dette gør vi ved at konfigurere følgende protokoller. Disse protokoller bliver ikke uddybet, da vi har valgt at gå helt i dybden med et udvalg af protokoller, men opsætningen er vedlagt i bilag.

1. IP adresse på interfaces
2. Security levels på ASA
3. Username, Passwords, Hostname og SSH
4. Gateways
5. Default routes
6. NAT
7. VLAN
8. Trunks og Access porte
9. IP Helpers
10. OSPF
11. STP
12. Etherchannel

### Netværksprotokoller:

I fitness junkies har vi valgt at uddybe følgende protokoller og forklare noget mere om da vi syntes de er væsentlige for opsætningen.

#### FHRP:

First hop redundancy protocol er en protocol der sørger for at netværket er højt tilgængeligt. Der findes 3 forskellige typer af FHRP. I fitness junkies har vi brugt HSRP som FHRP, da vi har arbejdet meget med den, og da den opfylder formålet med høj tilgængelighed.

#### HSRP:

HSRP også kaldt hot standby router protocol er en cisco proprietær protokol som sikre os høj tilgængelighed i netværket. HSRP består af en virtuel router som har en IP og MAC adresse. Der findes en active router og en standby router. Den active router betjener den virtuelle router og hvis den active router skulle ske at fejle så tager standby routeren over som active router.

I nogle setups ville der også være mere end 2 routere i en HSRP gruppe, hvis en tredje router skulle komme på ville den tjene som en Listen router, og være en backup for standby routeren.

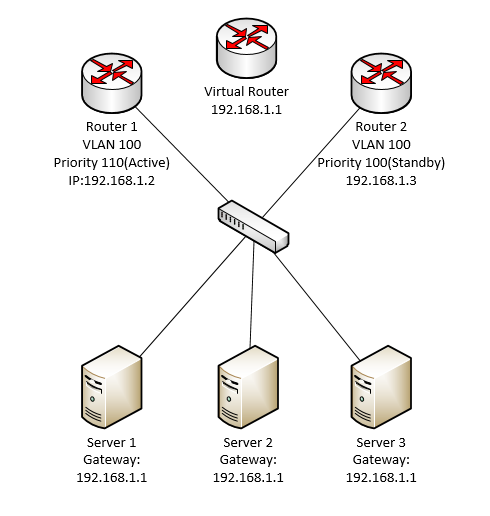
En listen router ville blive til standby router, hvis den aktive router skulle ske at slukke.

For at finde ud af, hvem er active, standby og listen sætter vi en prioritet på vores routere. Prioriteten er fra 0-255(Default 100), hvor den højeste prioritet vinder valget.

Hvis man ikke har sat en prioritet på sine routere og dermed beholder default på 100 vil

routeren med den højeste ip adresse blive til active og næste højest standby.

På billedet ser vi et eksempel på et typisk HSRP setup. Her er router 1 active for Vlan 100, da den har højere prioritet, hvilket betyder router 1 betjener den virtuelle router. Derfor kan vores servere sætte den virtuelle router som gateway.



##### Preempt featuren:

Når router 1 slukker bliver router 2 active router, men hvis router 1 så tænder igen, hvordan skal router 2 så vide den skal være standby igen fordi den har lavere prioritet? Til det har man lavet preempt featuren. Preempt featuren sørger for at den router med højest prioritet altid er active routeren, da den sender en coup message over til den active router som har lavest prioritet og siger “hej router 2 jeg er altså den active router da jeg har højest prioritet”.

##### Track featuren:

Lad os sige vi havde 20 vlans på router 1 og 2,hvis så Vlan 100 går ned på router 1 og routeren stadig kører videre, ville man kunne bruge track featuren til at decrement prioriteten med 20 på Vlan100, og dermed ville router 2 have højere prioritet end router 1 og dermed ville router 2 blive active på vlan 100.

##### Hello timers:

Hello timers bliver brugt imellem routerne for fx at finde ud af om de er nede. Svarer routeren ikke på hello pakken og der går mere end 10 sek, antager den at den er nede.

Hello pakker bliver default sendt hver 3 sekund og har en holdtime på 10 sek default. Holdtime er den tid det tager for routeren at finde ud af om den anden router er nede.

##### HSRP Versions:

Der findes to versioner af HSRP. Version 1 og Version 2.

Version 1 er default versionen som vi har konfigureret i vores miljø. Version 1 bruger multicast adressen 224.0.0.2 til at sende hello pakker.

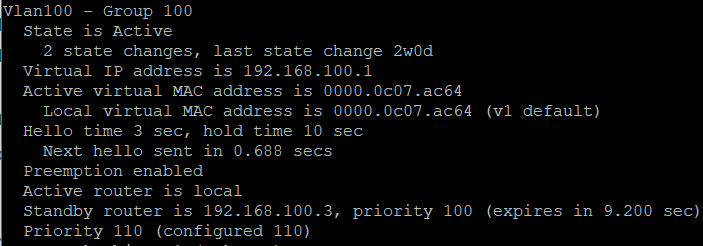
HSRP version 2 supportere IPv6 og bruger multicast adressen 224.0.0.102 til at sende hello pakker. De to versioner kører to forskellige formater og, man kan derfor ikke kører version 1 og 2 på samme tid.

##### Konfiguration:

I Fitness Junkies har vi sat HSRP op på vores VLAN. Her tager vi et eksempel ud fra vores server vlan på vores 2 distributions switche(FJ-Core01,FJ-Core02)

1. Vi starter med sætte vores virtuelle routers IP og et gruppe nummer. Gruppe nummeret er lokalt og efter cisco best practice giver vi gruppe nummeret samme nummer som vlan nummeret.
2. Standby 100 ip 192.168.100.1
3. Derefter sætter vi vores prioritet
4. Standby 100 priority 110
5. Så sætter vi preempt featuren på
6. Standby 100 preempt
7. Vi sætter track på, hvis Vlan100 skulle gå ned så decrementer den prioriten med 20 så vores anden router tager over for vlan 100
8. Standby 100 track 1 decrement 20
9. Derefter sætter vi samme konfiguration op på den anden switch:
10. Standby 100 ip 192.168.100.1
11. Vi behøver sætter ingen prioritet, dvs vi får en prioritet på 100(Default)
12. Preempt
13. Vi behøver ikke track da standby routeren har laveste prioritet.

Altså slukker FJ-Core01 decremter den prioriteten til 90, hvilket betyder FJ-Core02 bliver active router. Tænder FJ-Core01 igen får den prioriteten på 110 igen og pga preempt featuren bliver den active router igen. Bruger vi kommandoen “Show Standby”, så kan vi se, hvilken router der er active og standby.



På billedet over har vi en show standby fra FJ-Core01. Her kan vi se:

-Den er active

-192.168.100.3 er standby med prioriteten 100

-Den virtuelle IP og MAC  
-Hello timers, og hvornår den sender dem.

#### VRRP:

Som alternativ til HSRP kan man bruge VRRP - Virtual Router Redundancy Protocol, som næsten er samme protokol som HSRP. Fordelen ved VRRP er at det er en open standard, hvilket vil sige den fungere på alle enheder. Til forskel fra HSRP referere man active router som master router og standby som backup routers. I VRRP kan man have flere backup routers.

#### GLBP:

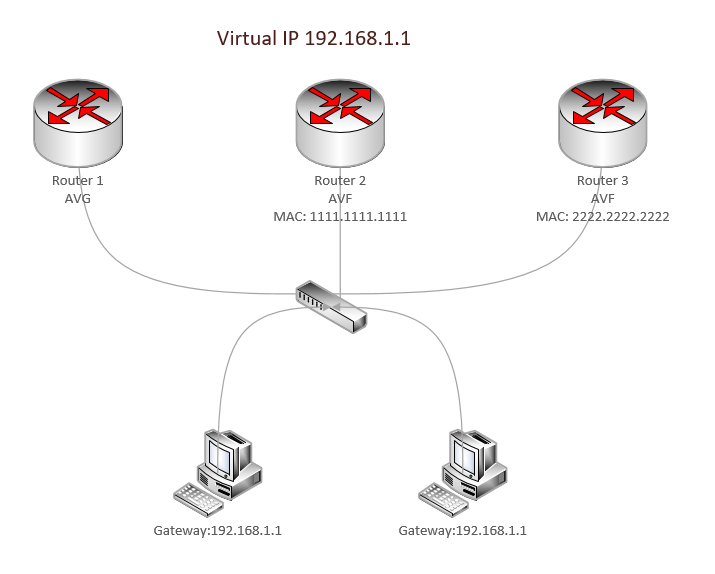
Gateway Load Balancing Protocol er anderledes. I HSRP og VRRP kan man kun have 1 router der forwarder trafik fra X vlan eller interface ad gangen. Med GLBP kan man forwarde trafikken på flere routere ad gangen. I GLBP har man en AVG(Active Virtual Gateway), der svarer på ARP Requests. Derudover kan man også have op til 4 aktive AVF(Active Virtual Forwarder) der kan sende trafik på samme tid. En AVF er et medlem af AVG der finder AVG ved hjælp af hello messages og lære den virtuelle MAC adresse fra AVG.

På billedet ser vi et typisk eksempel på GLBP. PC1 sender en ARP request til sin gateway for at finde ud af hvad mac adressen er. Router 1 svarer så tilbage også siger at den kan bruge Router 2 MAC adresse som er 1111.1111.1111, og dermed vil PC1 bruge router 2 når den sender pakker til sin default gateway. Når PC2 så gør det samme og sender en ARP request vil router 1 svarer tilbage og sige den kan bruge Router 3 MAC altså 2222.2222.2222. Der findes 3 former for load balance i GLBP:

Round-Robin: Den skifter fra avf til avf, så det hele er lige fordelt(Default)

Host-dependent: Her kan få en host til at få den samme MAC adresse hver gang fra AVG.

Weighted: Her kan man sætte en vægt så den bruger den ene router mere end den anden. Det vil man typisk bruge, hvis den ene router er kraftiger end den anden.



FHRP på ASA:

På ASA 5505 kan man ikke kører HSRP, de kan til gengæld kører noget der hedder “Failover”. I vores proof of concept kunne jeg ikke helt få det til at virke. Jeg vil dog gerne forklare, hvad jeg kom frem til. I failover sætter man et primary link på den ene asa og et secondary link på den anden ASA, og dermed kan man lave et interface designeret til failover.

Da en ASA 5505 ikke har routed interfaces, gjorde jeg det på Vlan. Først angiver jeg den ene asa som primary og den anden som secondary med følgende kommando:

#failover lan unit primary

#failover lan unit secondary

Derefter laver jeg interfaced og tilslutter det til et Vlan:

#failover lan interface Failover Vlan40

Vi angiver nu en ip adresse og en standby adresse:

#failover interfaces ip Failover 192.168.1.2 255.255.255.0 standby 192.168.1.1

Vi angiver en secret key som skal være ens i begge ender

#failover key Password1

Vi kører nu samme konfiguration op i den anden ende.

Derefter skulle man efter cisco kunne konfigurere standby op på sit inside og outside Vlan. Selve failover linket jeg har lavet virker fint nok, men jeg kunne ikke pinge den virtuelle adresse på mit outside og inside interface som jeg havde lavet. Derfor valgte jeg kun at kører med 1 ASA.

## Port security:

I fitness junkie har vi sørget for port security. Der findes et hav af forskellige port security protokoller, men vi har pillet dem ud som vi synes er væsentlige for virksomheden og implementeret dem i netværket.

### Maximum MAC:

I fitness junkies har vi sat maximum 5 MAC adresser på hver port i netværket. Vi gør det af ren sikkerhed. En switch har en MAC adresse tabel som maximum kan håndtere et hvis antal af MAC adresser ad gangen. Hvis en hacker sender en hel masse pakker ud til switchen med en forskellig source MAC i hver pakke, så ville tabellen bliver overfyldt og switchen kunne risikere at gå ned eller hackeren kunne sætte en Wireshark på og opsamle vigtige pakker. Man kan fx lave disse angreb via en Linux distribution som har programmer til det. Vi konfigurere maximum 5 MAC adresser med kommandoen:

#switchport port-security maximum 5

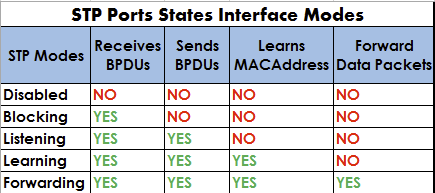
### Portfast:

Portfast gør at switch eller trunk porten går direkte ind i forwarding state, ved at springe listening og learning starte over som tager 30 sekunder. Det vil betyde at når man sætter en pc til switchen, skal man ikke vente 30 sekunder på at porten kommer op, fordi den skal igennem listening og learning. Der findes 5 forskellige states.

Listening state tager 15 sekunder, hvor den lytter efter BPDU.

BPDU er pakker der bliver sendt frem og tilbage imellem switchene der indeholder information på porte, prioriteter og cost, for at sørge for der ikke opstår loops i netværket.

Learning state tager også 15 sekunder, men her lære den også MAC adressen.



Da vi ved vi sætter computere til vores portfast porte og ikke switche, kan vi med tryghed sætte portfast op på de porte der bliver brugt til klienterne. Dette gør vi ved at skrive kommandoen:

#spanning-tree portfast

Hvis der nu skulle blive sat en switch til en Portfast port kunne der opstå loop da den springer listen og forward state over. Derfor er det vigtigt vi sørge for at enable BPDU guard. BPDU guard sørger for at lave en shutdown på porten, hvis den skulle ske at modtage BPDU pakker. Man konfigurere BPDU guard ved at gå ind på sit interface og bruge kommandoen: spanning-tree bpduguard enable.

### Storm-Control:

Som tidligere har vi sat BPDU guard op på portene så eksterne switche ville blive blokeret, hvis de blev sat til netværket, men hvad hvis man sætter en switch til netværket som ikke sender eller modtager BPDU pakker som en dum switch, eller, hvis man sætter en managed switch til som ikke deltager i spanning-tree?

Hvis en bruger så satte en switch til netværket, ville den kunne tage netværket ned, hvis der opstod en broadcast storm. En broadcast storm opstår, hvis der sker en kortslutning i switchen. Dette vil jeg demonstrere i vores proof of concept.

Derfor kan man bruge Storm-Control protocol. Storm-Control lukker den angivne port ned i error disabled mode, hvis den bruger et X antal af portens kapacitet på broadcast. Derudover sender den også en SNMP trap at porten er lukket ned, og hvorfor. I fitness Junkies har vi konfigureret storm-control på følgende måde:

#storm-control broadcast level 10.00

#storm-control action shutdown

#storm-control action trap

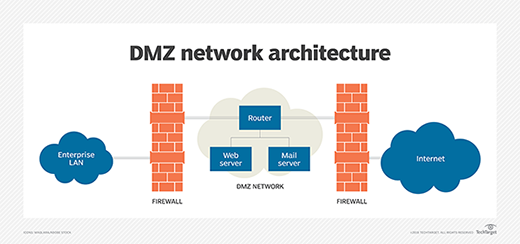
Når porten er gået i error disabled state oplever jeg at jeg ikke kan få porten op og kører igen med en no shutdown. Derfor har jeg brugt disse to kommandoer som gør at når porten går i error disabled mode på grund af broadcast storm så går porten automatisk op igen efter 30 sekunder, hvis broadcast stormen er stoppet.

errdisable recovery cause storm-control

errdisable recovery interval 30

## DMZ:

DMZ demoralized zone er en vigtig del af vores setup sikkerhedsmæssigt, da vi har en app som skal ramme en webserver vi skal hoste på netværket. Ved at lave en DMZ zone kan vi beskytte vores LAN, hvis en hacker skulle overtage kontrollen af vores web server.



### Opsætning:

I fitness junkie laver vi dmz zonen i firewallen da ciscos ASA firewall tilbyder en DMZ feature. I en asa kan man lave forskellige “security levels”. Med security levels kan man differentiere, hvor trafikken er tilladt at komme hen. Et security level bliver defineret fra 0-100. Det er ikke muligt at komme fra et lavere security level til et højere security level uden at lave access regler i firewallen som tillader det. På den måde kan vi nemt blokere trafik.

Trafik fra et højere security level til et lavere security level er fuld tilladt, med mindre der er lavet access regler. Det vil sige vi kan sætte et eksempel op.

Typisk sætter man sit outside security level til 0 fordi man selvfølgelig ikke stoler på trafik udefra. Det vil sige alt trafik udefra og ind i netværket bliver blokeret. Derefter sætter man sit security level på sit Lan til 100, fordi man gerne vil tillade alt trafik fra sit Lan og ud imod lavere security levels altså internettet.

For at lave en DMZ zone sætter man sit security level imellem det security level man har angivet på sit outside og lan interface. Dermed får man en DMZ zone. Trafikken som er tilladt og blokkeret ville så sådan ud:

LAN>DMZ Trafik Tilladt

LAN>Outside Trafik tilladt

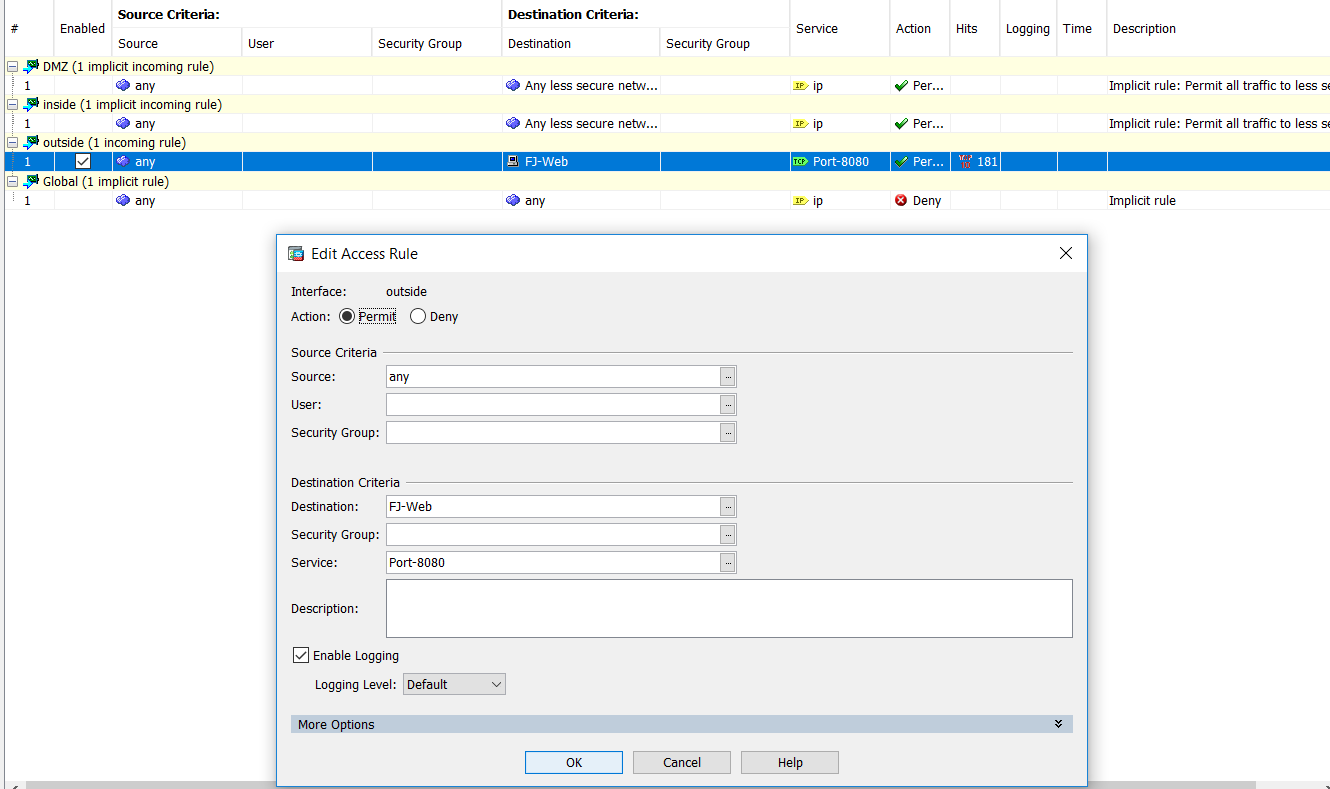
DMZ>LAN Blokkeret

Outside>Lan Blokeret

Outside>DMZ Blokeret

Derefter laver vi en regel som tillader http trafik fra outside til dmz på port 80. For en sikkerheds skyld laver vi port 80 om til 8080 når den rammer web serveren.

Først tillader vi trafik hen til web-serveren på port 8080.



Derefter laver vi en port forward, der gør trafik udefra bliver lavet om fra port 80 til 8080 når det skal hen til Web-serveren.

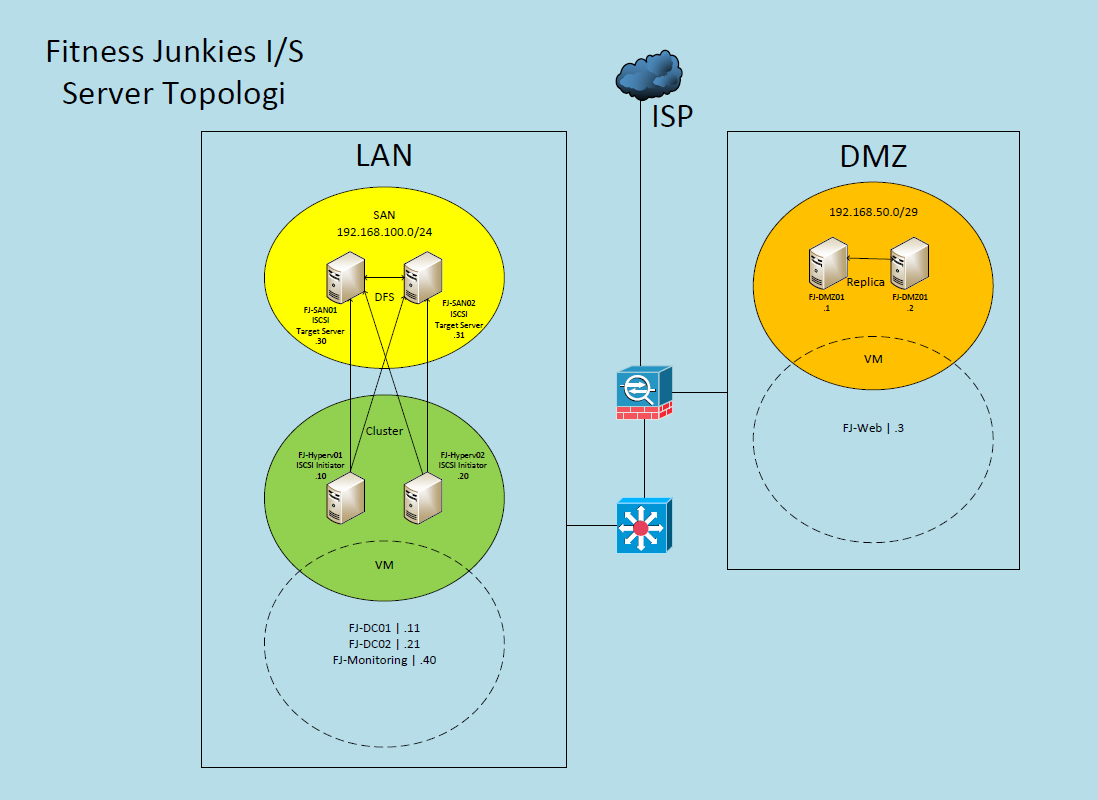


Vi kan se på reglen under translated packet at trafik fra outside til web-serveren bliver translated fra port 80 til 8080.

# Opstart af fitness Junkies Server Drift

Vi vil gerne bygges vores bedste bud på et fuld funktionelt Server miljø med design, tilgængelighed, sikkerhed og skalerbarhed i fokus, så appen altid er kørende.

### Server topologi(Topologi vedhæft i bilag så man kan zoom):



### Begrænsning:

Da vi ikke har nok diske i maskinerne kan vi ikke opsætte Raid i clusteret i proof of concept løsningen. Raid-5 ville være optimalt så hvis en disk dør i SAN så mister brugerne ikke alt deres data på den pågældende disk.

### Valg af udstyr

2x SAN Servere bestående af Windows Server 2016.

4x Hyperv Servere bestående af Windows Server 2016

Virtuelle maskiner:

2x Domain controllere Bestående af Windows Server 2016

1x Web server samt database til App bestående af en Ubuntu 18 Server

1x Monitoring server til overvågning bestående af en Ubuntu 18 Server

## Software, Services samt valg og opsætning af dem:

### Windows Server 2016 Datacenter:

Vi har har valgt at bruge Windows Server 2016 Datacenter til opsætning af SAN, DFS, Cluster og domain. Vi vælger at bruge Windows server 2016, da det er det styresystem vi har mest kendskab til og da det tilegner sig appens funktionalitet og virksomhedens krav. Vi har valgt ikke at bruge det nye Windows Server 2019, da vi ikke har så meget kendskab til styresystemet endnu og da det er helt nyt og sikkert har nogle fejl i sig.

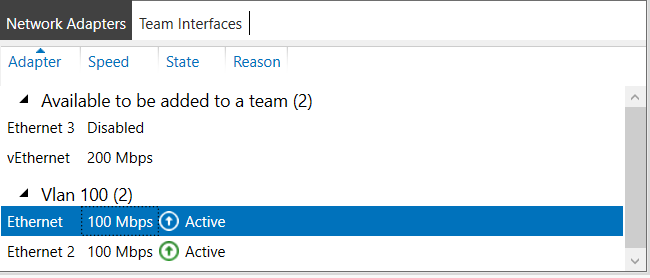
### Installation:

Windows Server 2016 Datacenter er installeret på 6 maskiner. Maskinerne er derefter blevet sat op med en grundkonfiguration, hvilket vil sige NIC Teaming(på de 4 hyper-v maskiner), Servernavn og remote desktop, hvor de efterfølgende er klar til at blive konfigureret.

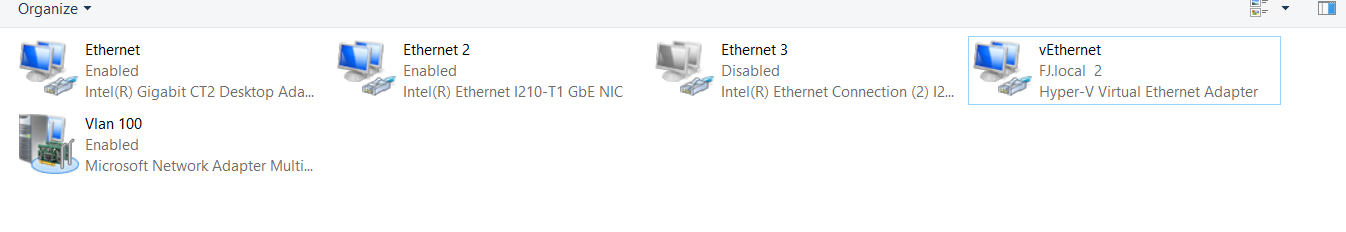
### NIC Teaming Opsætning:

Krav: 2x+ Netværkskort

NIC teaming gør det muligt at lave redundans på på servernes netværks link, hvilket vil sige at, hvis switchen serveren er koblet til dør så har den et andet link over til en anden switch som kører. Dette gør det muligt for serveren at være høj tilgængelig. NIC Teaming sætter man op via server manageren, hvor man laver et team og bruger “switch independent link” funktionen, hvilket betyder at linket er uafhængig af switchen. Hvis man ikke brugte den feature ville man skulle konfigurere en etherchannel protokol på switchen som ex LACP. Her ser vi to netværkskort der er tilføjet til et NIC Team ved navn Vlan 100.



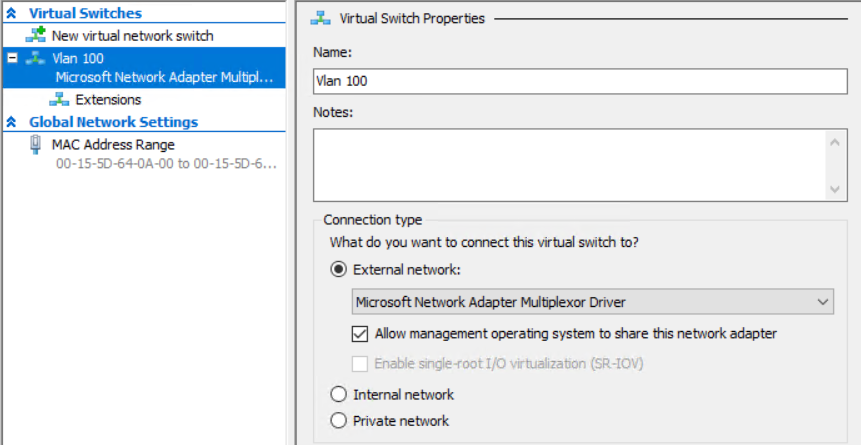
Når man går ind i netværkskort på computeren vil NIC teamet være tilføjet ved navn Vlan 100 ved navn Microsoft Network Adapter Multiplexor driver. Dette kort skal bruges til opsætning af det virtuelle netværks kort til hyper-v som jeg vil forklarer i Hyper-v Sektionen.

Hyper-v:

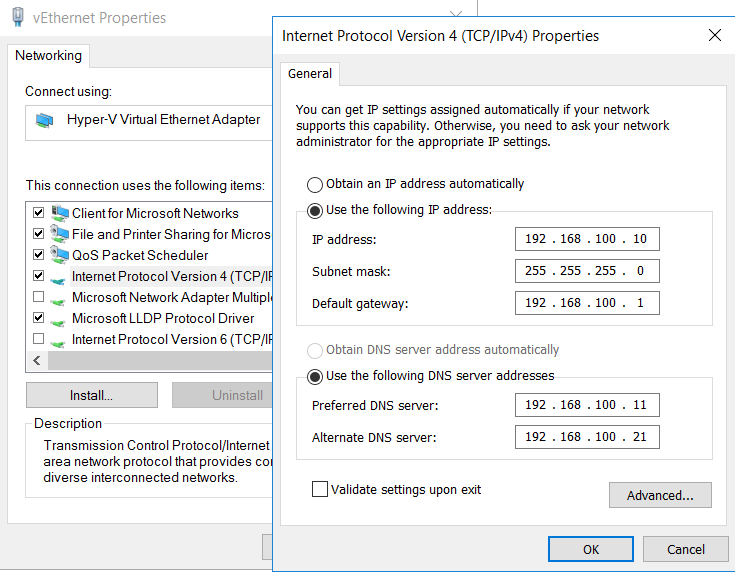
Vi har valgt at bruge Hyper-v som hypervisor. En hypervisor er den maskine som styre de virtuelle maskiner. Hyper-v har vi arbejdet med i mange år og har stor kendskab til. Hyper-v tilegner sig også de services som vi sætter op i miljøet, som blandt andet replikering. Man kunne også bruge software som ESXI(Vmware), hvilket også kan udføre samme job. ESXI har også mange gode egenskaber, men kræver licenser for at kunne udføre X jobs. I vores proof of concept er hyper-v en god løsning da man kan få fuld funktionalitet i en trial version.

### Opsætning af Hyper-v

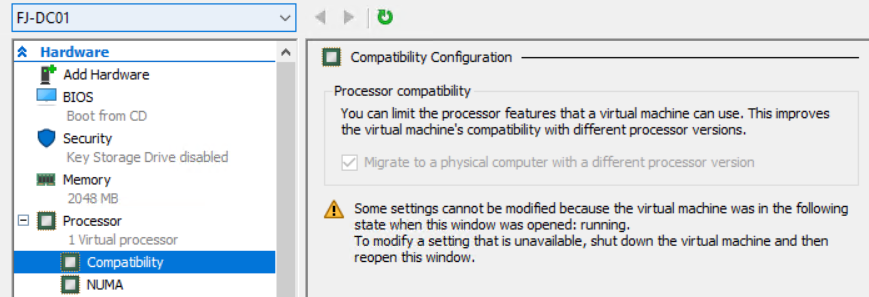
Som det første installere vi Hyper-v servicen i server manager på de 4 maskiner som skal bruge det. Efter Hyper-v er installeret er vi klar til at opsætte en virtual switch. Det gør vi ved at vælge virtual switch manager og oprette en virtual switch og tilføje vores netværkskort vi lavede i NIC teaming(Vlan 100,Multiplexor driver). Vi laver 4 virtuelle switche i alt 2 til Vlan100 og 2 til Vlan50(DMZ)



Derefter vil vi opleve at den virtuelle switch er blevet aktiv og nu kan vi sætte en ip adresse på vores hyper-v maskiner inde i det virtuelle netværkskort.



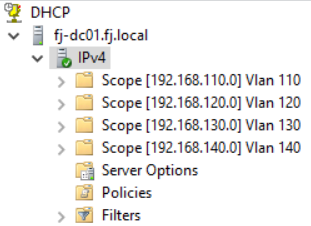
Nu har vi hul igennem netværket og er klar til at sætte VM op.  
I fitness junkies er der oprettet 4 virtuelle maskiner bestående af 2 domain controllere, 1 monitoring og Web server. Når vi opretter en VM er det vigtigt at vi laver en virtual disk der er gemt i cluster drevet så vi får høj tilgængelighed. Til at starte med oprettet jeg de virtuelle maskiner på det lokale C: drev hvilket resulteret i jeg skulle flytte alle de virtuelle disks over på Clusteret, hvilket tager en del tid i proof of concept miljøet da overførselshastigheden ikke er særlig hurtig. Når maskinen er oprettet skal vi sørge for at sætte flueben Migrate som vist på billedet, da vores to hyper-v maskiner har to forskellige CPU typer. Hvis vi ikke sætter flueben i den vil det resultere i at maskinen ikke kan migrate, hvis en failure opstår.



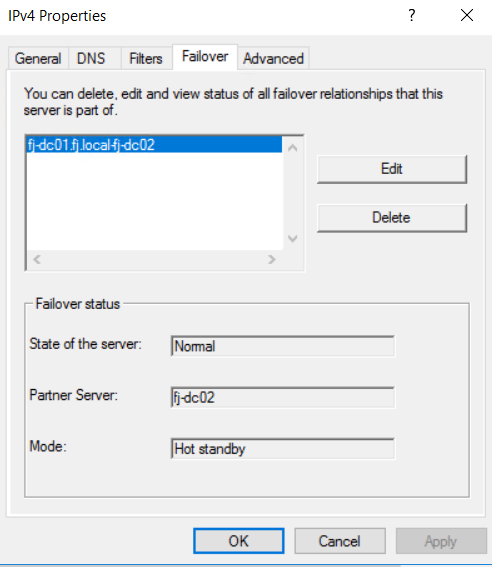
### Opsætning af Domain Controllere:

Vi opsætter 2x domain controllere for redundans så der altid er en domain controller der kører. Vores domain controllere består af FJ-DC01 og FJ-DC02. Vi installere AD Domain Services på begge servere. Opretter en ny forrest “FJ.Local” på den ene server og joiner en existing forrest på den anden server. På den måde får vi to domain controllere. Når domain controllerne er lavet joiner vi domænet på alle vores server undtagen de servere som er DMZ zonen, de skal være isoleret fra domænet da det er et security breach.

Derefter installere vi DHCP services på begge servere, går ind på den ene server og opretter DHCP scopes til virksomheden så de ansatte får en IP adresse på netværket.



Derefter laver vi en failover scope så, hvis den ene server går ned så er der stadig en fungerende DHCP server. Vi højreklikker på IPv4 og vælger configure Failover. Vi tilføjer den anden domain controller og sætter den som hot standby. Vi kan derefter se inde under egenskaber at failover er aktiv og at fj-dc02 er hot standby.



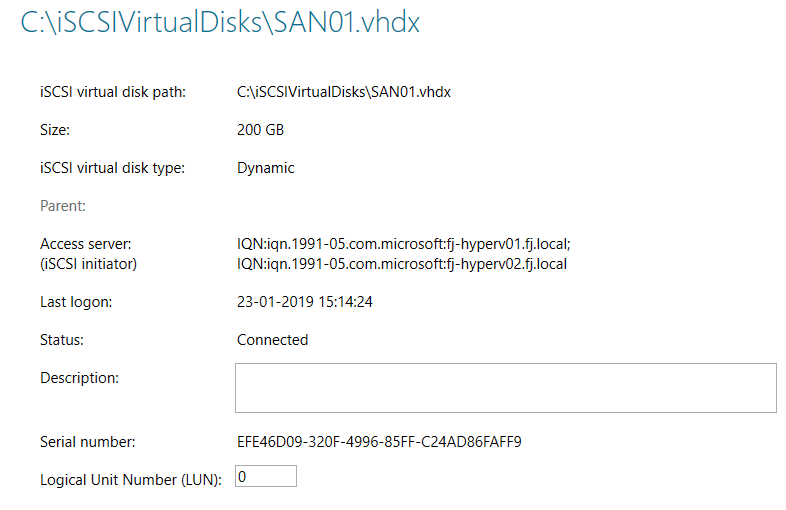
### Fejlfinding:

I starten havde jeg kun 1 domain controller, hvilket resulteret i at der den hyper-v server slukket som domain controlleren kørte på, så døde clusteret, fordi clusteret bruger DNS til at holde det i live, og da domain controllerens virtuelle disk ligger på clusteret, så kan den virtuelle maskine ikke boote igen fordi clustered er død, også lukkede jeg mig selv helt ud af systemet. Derfor bruger jeg to domain controllere og peger på begge domain controllere i IP opsætningen.

### Opsætning af Windows Cluster:

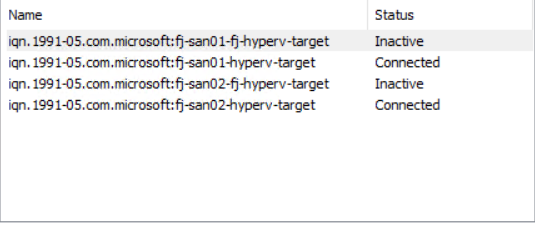
Hvorfor Cluster? Cluster er en god feature Microsoft har lavet. Formålet er at gøre server miljøet højt tilgængeligt så det aldrig går ned. Hvis en hyper-v server skulle ske at slukke af en eller anden årsag, så bliver alle maskinerne migreret over på den anden Hyper-v maskine via Live Migration som er en migration der foregår lynhurtigt uden brugerne enlig oplever downtime. Cluster har også mange andre funktioner som jeg vil forklarer om senere.

Vores cluster setup består af 4 maskiner:  
FJ-SAN01,FJ-SAN02, FJ-Hyperv01 og FJ-Hyperv02. Vi starter med at konfigurere ISCSI som er en protokol der gør vi kan overføre data med TCP i et SAN(Storage Area Network), hvilket vil sige vores data data kan deles over netværket. Dette gør vi kan få høj tilgængelig data. Først installere vi servicen ISCSI target på begge servere. Et ISCSI target er der, hvor vi har vores diske og opbevare data på. I et ISCSI target miljø sætter man typisk et RAID-5 eller 6 miljø op som gør at, hvis en disk skulle fejle så har man stripet data’en ud på de andre diske med paritet, som gør at man kan miste en disk uden at miste alt sit data, men da jeg ikke har nok diske til rådighed så kan jeg ikke lave dette setup. Disk striping with parity kan både laves i software i disk manageren eller via en hardware controller som man typisk gør fordi det er hurtigere. Når Target servicen er installeret går vi ind i server manager og opretter en shared disk inde under ISCSI som peger på de to Hyper-v servere som er initiators. Her har oprettet den første disk på SAN og tildelt den fj-hyperv01 og fj-hyperv02 som initiator.



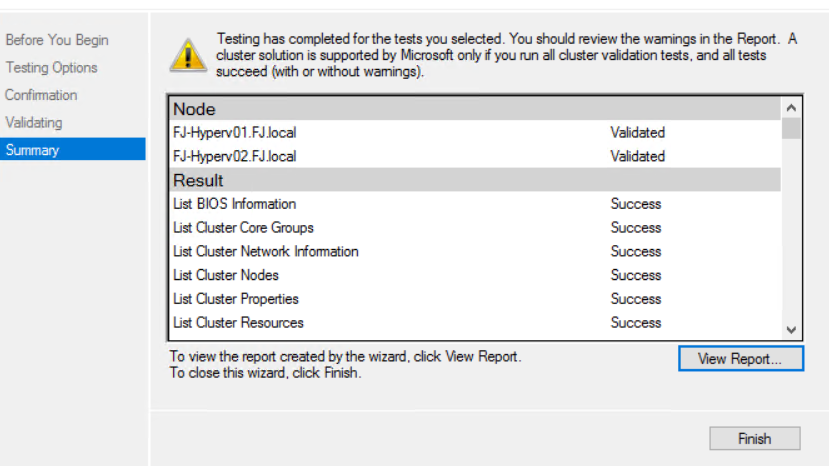
En ISCSI initiator er dem som bruger disken der er delt. Vi har oprettet 2 disks på hver server så vi i alt har 4 disks bestående af SAN01,SAN02,SAN03 og SAN04.

Derefter går vi ind på hyper-v serverne og initator vores diske. Vi initiator diskene på san01 og san02 på begge hyper-v servere. De 2 Inactive sessions på billedet er nogen der ikke bliver brugt længere.



Nu kan vi gå ind i en vilkårlig hyper-v server i disk management og bringe diskene online og formatere dem med NTFS.

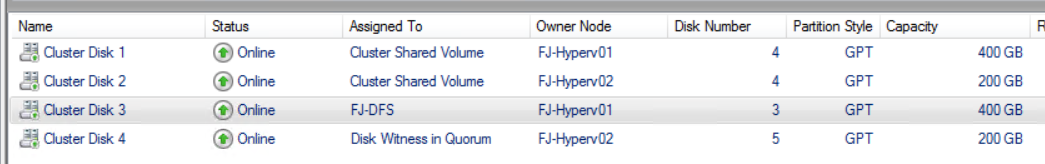
Nu er klar til at konfigurere failover clustering. Vi starter med at installere failover clustering servicen på begge hyper-v servere. Derefter åbner vi cluster manager på en vilkårlig server og kører en cluster validation. En cluster validation sørger for at clusteret er klar til at blive oprettet og at der ingen fejl i pre konfigurationen, som kunne være netværket eller hyper-v konfigurationen. I den sidste validation jeg har kørt får jeg en warning på jeg skal være sikker på at mine virtuelle maskiner er startet op, men det er de. I nogle tilfælde ville der opstå errors i validation som er vigtige at få rettet inden man laver clusteret ellers vil det ikke fungere optimalt.



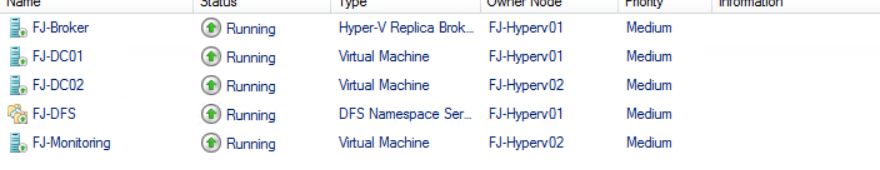
I starten døjede jeg med mine hyper-v switche fordi de ikke hed det samme på begge hyper-v servere. Jeg kunne derfor ikke få lov til at oprette clusteret, hvilket er underligt, og kunne simpelthen ikke finde ud af, hvorfor de to virtuelle switche skulle hedde det samme for at få det til at fungere.

Efterfølgende er vi klar til at lave vores cluster. Vi højre klikker på failover cluster og vælger create new cluster.

Vi giver nu Clusteret et navn, IP adresse og tilføjer de 4 disks som vi har lavet. Nu vil clusteret være oppe, og det vil være muligt at se at vi har failover på vores disks ved at se owner og at de skifter owner når den ene server slukker. Vi kan også højre klikke på disken og trykke på move og flytte disk owneren fra den ene hyper-v maskine til den anden. Man vil bemærke at en af diskene hedder Disk Witness in Quorum. Denne disk opbevare database konfigurationen over clusteret som man også kalder quorum. Denne disk sørger for at fortælle hvilken en af hyper-v serverne der skal være aktiv. Der vil også være en disk der er assigned til DFS. Dette vil jeg forklare mere om i DFS sektionen. De to resterende disks skal bruges til at opbevare Virtuelle disks på til VM.



Nu vil vi gerne gøre det muligt at migrere vores virtuelle maskiner over på en anden maskine, hvis en hyper-v maskine skulle ske at slukke. Først højre klikker vi på role og vælger rollen hyper-v broker også kaldet Client Access Point og giver den et navn og en IP adresse. En Hyper-v broker er en virtuel replica server som gør det muligt at replikere vores VM fra server til server, hvis en maskine skulle gå ned. Nu kan vi tilføje vores VM ved at vælge roles og vælge virtual machines og tilføje dem. Derefter vil den sætte dem op og se om der er nogen fejl. En typisk fejl ville være at man ikke har gemt enten den virtuelle disk eller den virtuelle maskine på de shared diske.

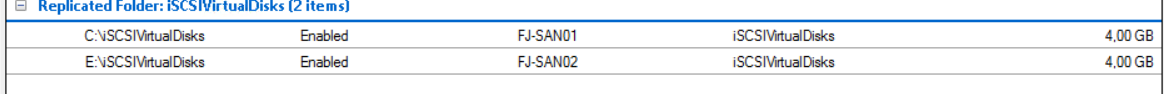


Nu er vores VM highly available og det kan vi teste ved at slukke den ene server og se alle servere bliver live migreret over på den anden server. eller højre klikke på VM og flytte den til den anden node via live migration, som jeg vil vise i produkt rapporten.

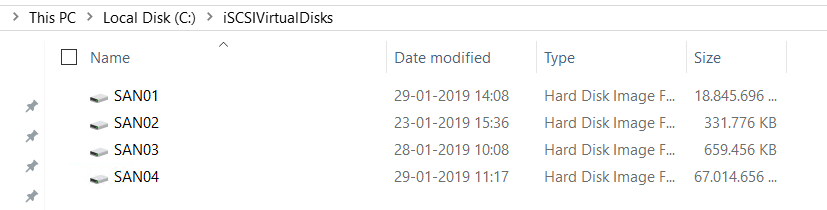
### 

### DFS:

Distributed file system gør det muligt at replikere data mellem to eller flere servere for høj tilgængelighed. Vi bruger DFS imellem vores SAN servere for at lave en kopi af vores SAN diske, så hvis diskene skulle ske at dø så har vi samme data på den anden server. Normalt ville man bruge RAID til denne funktion men da vi ikke har nok diske til rådighed har vi valgt at køre DFS replikering imellem de to servere. Vi starter med at installere DFS Service på begge SAN servere. Åbner DFS management og går ind under replikering. Vi starter en ny replikering, tilføjer de to SAN servere og de to mapper, hvor SAN diskene ligger i, hvorefter vi peger dem imod hinanden så de replikere, som vist på billedet.

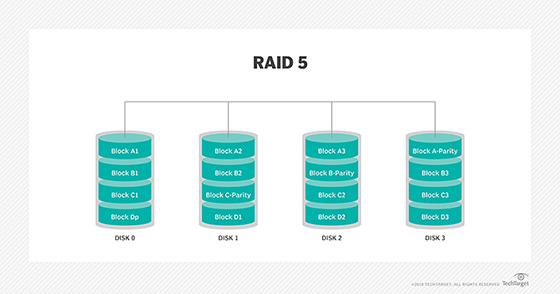


Derefter sætter man et replica schedule op som kører en gang i døgnet, hvor man replikere alle filerne over på den anden server. Efter replikationen er kørt kan man se at diskene er flyttet over på den anden server.



Der opstår bare endnu et problem med denne løsning. Lad os sige 1 SAN Server dør. Så mister vi to diske og vores ISCSI initiators som peger på serveren ville gå offline og clusteret ville miste 2 diske. Vi kan lave en lille work around, ved at importere de replikeret SAN diske på den server som stadig kører, men det vil kræve en manual handling hvis et SAN server går ned. Derfor vil RAID være det mest optimale.

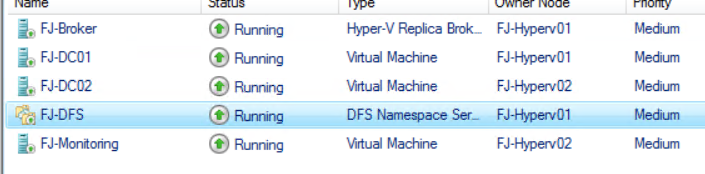
I vores rigtige løsning som ikke er proof of concept vil vi sætte et RAID-5 miljø op. I et RAID-5 miljø ville vi ikke stå med dette problem da vi har redundans på vores diske. I et RAID 5 miljø, ville man kunne skifte den defekte disk ud og sætte en ny i med et såkaldt “hot Swap”, hvilket betyder at man kan gøre det imens systemet er kørende. RAID 5 består af block striping med paritet fordi alle data og pariteter er striped ud på alle diske. Når data bliver skrevet ud til RAID 5 drevet, beregner systemet pariteten og skriver det ind i diskene, imens spejling opretholder flere kopier af data i hvert volume. Dette vil resultere i, hvis vi mister en disk så har vi stadig alt vores data. I et RAID 5 miljø er man nødt til at have +3 diske. På billedet ser vi et eksempel på et RAID 5 setup. Hvis vi fx mister disk 2 så vil de 3 resterende diske have en kopi af disk 2, hvilket gør vi kan hot swap disken. Hvis vi fx mister 2 diske på samme tid ville RAID 5 systemet dø da vi kun har 1 paritet på hver disk. Med Raid 6 kunne man løse dette ved at have 2 pariteter på hver disk, så vil systemet kunne holde til 2 disk failures.



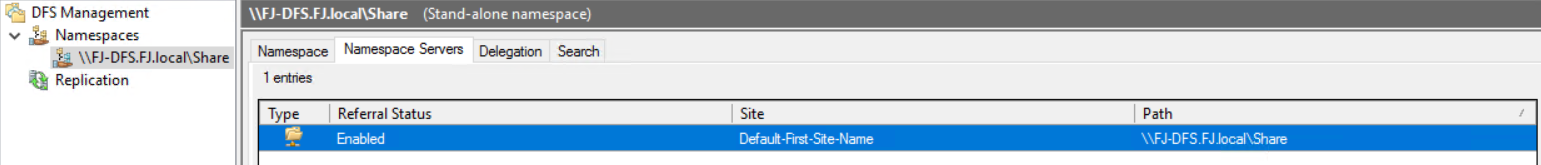
### DFS på Fileshare:

Vi vil også gerne bruge DFS på vores fileshare til fitness junkies ansatte. Dette vil gøre at alt det de ansatte gemmer på deres drev altid er højt tilgængeligt så de kan arbejde videre hvis en hyper-v server skulle ske at slukke.

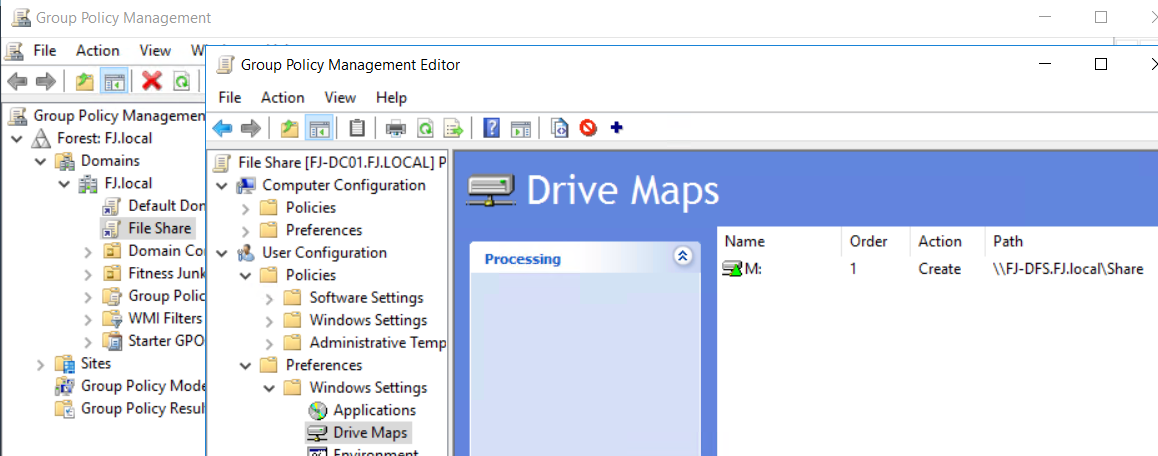
Vi starter med at installere DFS servicen på begge Hyper-v servere. Da vi har sat et cluster op er vi nød til at lave DFS via roles i clusteret, da vi laver det på en cluster disk. I starten prøvede jeg at lave DFS igennem server manager, hvilket resulteret i at jeg fik en error. Derfor er det vigtigt vi vælger roles og tilføjer DFS Server og giver den en IP adresse.



Nu kan vi højre klikke på DFS og vælge management og oprette et Namespace som vi kalder for share. Vi starter med at tilføje vores to Hyper-v servere og laver mappen Share på vores SAN Volume.



Et namespace er en shared folder som begge hyper-v servere hoster, hvilket gør at hvis den ene server slukker så hoster den anden server den shared folder. Derefter kan vi tilføje rettigheder til mappen så de ansatte kan redigere i filerne. Nu kan vi gå ind på domain controlleren og åbne Group policy management og tilføje mappen. Så den bliver mappet til alle ansatte.



### Backup:

I Fitness Junkies har vi taget hensyn til disaster recovery, hvilket vil sige at, hvis bygningen skulle ske at springe i luften eller bliver oversvømmet så har vi lavet full backup på et eksternt medie 1 gang om ugen med Windows Server Backup. Der vil blive gemt full backup så man altid kan gå 3 måneder tilbage i tiden og recovery. Kommer man mere end 3 måneder tilbage vil det være slettet. På den måde har vi altid sikret dataen, hvis nu en katastrofe skulle ske. For at sætte backup op installere man først servicen Windows Server Backup på alle fysiske servere og derefter tilføjer et schedule der laver full backup, og peger det på det eksterne medie. I fitness junkies vil vi gerne tilbyde en QNAP, som er en god løsning til dette, da den kan stå i en anden bygning og kører backup så den ikke er samme fysiske sted, hvis en disaster skulle ske. I proof of concept løsningen vil QNAP ikke være en del af det, da vi ikke har en til rådighed.

### Opsætning af DMZ Zone og Web Server:

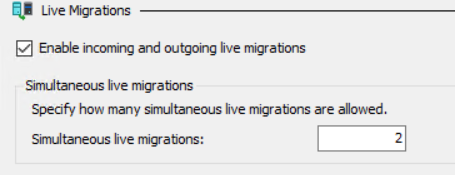
I Fitness Junkies er sikkerhed en top prioritet og derfor har vi lavet en DMZ Zone. DMZ zonen er beskrevet i netværks delen. Men som sagt er DMZ zonen en ekstra sikkerhed da vi blokkere alt trafik fra DMZ zonen til LAN. Hvilket gør at, hvis en hacker skulle ske at komme igennem netværket på port 80 og overtage serveren så ville han ikke kunne komme længere ind i netværket.

DMZ zonen består af:

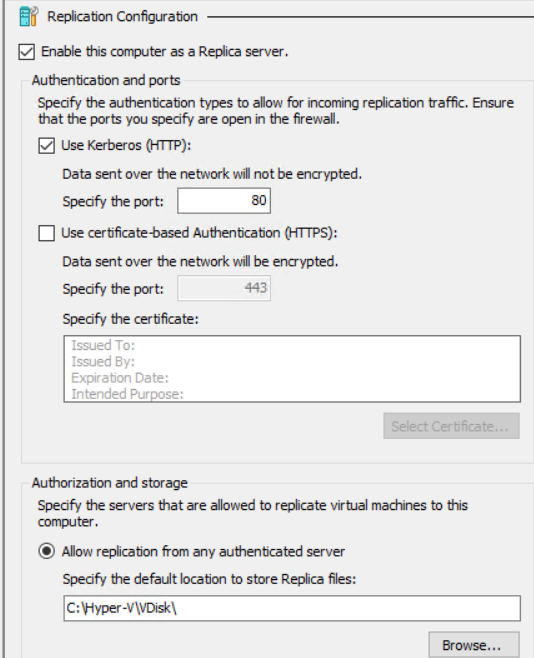
FJ-DMZ01, FJ-DMZ02 og FJ-Web.

I DMZ zonen har vi ingen cluster, da vi syntes det er lidt et overkill, da vi kun skal bruge 1 web server. Derfor har vi sat 2 hyper-v servere op med replica.

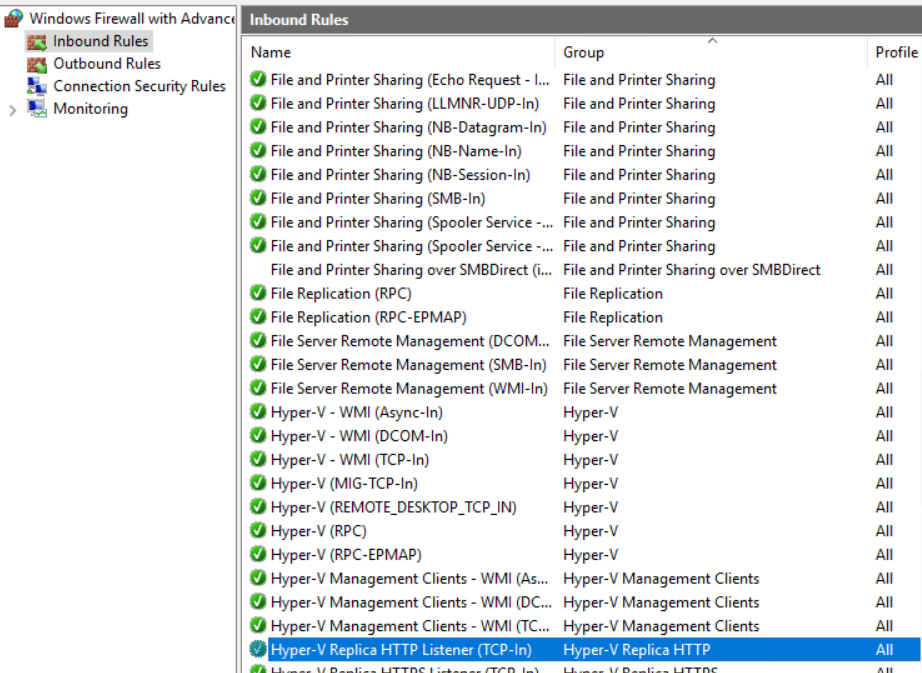
FJ-DMZ01 og FJ-DMZ02 på deres eget domæne DMZ.com. på disse to servere er der installeret Hyper-v. På serverne har vi sat Replikering op i hyper-v. Det har den samme funktion som i clusteret. Hvis den server med den hostet VM går ned så bliver den replikeret over på den anden server og kører videre. Man sætter det op ved at gå ind i hyper-v settings og enable live migrations.



Derefter sætter vi authentication op. Vi starter med at enable this computer as replica server. Derefter sætter vi replication trafikken op til at køre over port 80.

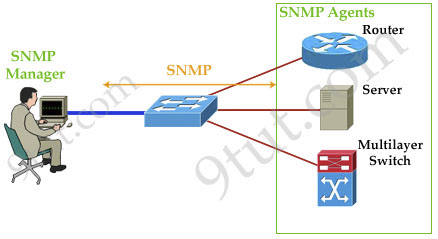


Det er meget vigtigt at allow replica trafikken i firewallen ellers kommer det ikke til at fungere. Havde problemer med dette til at begynde med.



### SNMP:

For at kunne overvåge serverne i Fitness Junkies har vi valgt at bruge en monitoring server. På den måde kan vi blandt andet se, hvad temperatur serverne og netværksudstyret har, hvor meget diskplads der er tilgængelig og hvor hårdt belastet netværket er. Til det formål vælger vi at bruge protokollen SNMP. SNMP bruger protokol nummer 161 og 162 til traps. Protokol 161 bliver brugt til at lytte til NMS fra agenten, og port 162 kommer fra agenten til NMS med en trap, hvis noget uforudset sker som fx en port shutdown på en router. SNMP består af en NMS også kaldt network management station som er den station/enhed vi styrer vores managed devices på.



De managed devices som kunne være en server, router eller en printer er der installeret et stykke software som hedder en SNMP agent, som reportere informationer til NMS. En SNMP agent er typisk installeret på det meste udstyr out of the box nu om dage. Agent softwaren består af forskellige OID numre som indeholder forskellige funktioner, som fx temperaturen på en router. Et OID nummer kunne fx være:

1 . 3 . 6 . 1 . 4 . 1 . 2682 . 1 . 4 . 5 . 1 . 1. 99 . 1 . 1 . 6

disse numre bliver så oversat i vores NMS så vi kan se det grafisk.

Derefter vil man kunne gå ind på sit device og enable SNMP og sætte en community string som er den string man skal bruge for at NMS kan kontakte den. Afhængig af, hvilken version man bruger af SNMP vil der være tilføjet flere lag af sikkerhed. Der findes 3 SNMP typer

Version 1: I SNMP version 1 kan man lave en 32 bit string som er plaintext. Hvilket vil sige alle kan se den og der vil ikke være nogen former for kryptering. Version 1 er en meget gammel protokol som ikke rigtig bliver brugt længere, men nogle få devices vil stadig bruge denne protokol. Skulle man ske et løbe ind i en version 1 så ville vi anbefale at sætte community string til read only, hvilket gør at maskinen kun kan læse data.

Version 2c: Version 2c fungere næsten på samme måde som version 1. Der er stadig ingen sikkerhed så read only er best practice. I version 2c er 64 strings supporteret, hvilket vil betyder at polling intervallet er hurtigere imellem NMS og device.

Version 3: I version 3 tilføjer vi sikkerheden som en ny feature her kan vi både bruge authentication altså username og password og encryption altså vi kan kryptere vores trafik imellem device og NMS.

I SNMP er det vigtigt at bruge samme version i begge ender ellers fungere kommunikationen imellem end det managed device og NMS ikke!

### Opsætning af Monitoring Server:

I vores miljø har vi valgt at bruge en Ubuntu 18 server som NMS. Vi har valgt at bruge LibreNMS da vi syntes at det er et velfungerende stykke software som viser præcis hvad vi ønsker. Derudover er softwaren også gratis. Vi starter med at installere Ubuntu 18 Server på en VM i DMZ miljøet. Vi kører maskinen op med basic konfiguration og opdatere den altså IP,DNS,OpenSSH, Apt-get update og upgrade Derefter installere vi LibreNMS. Vi husker at løfte brugeren til admin ved at skrive sudo bash og indtaste password.

Vi starter med at hente prerequisites som er følgende:

apt install curl apache2 composer fping git graphviz imagemagick libapache2-mod-php7.2 mariadb-client mariadb-server mtr-tiny nmap php7.2-cli php7.2-curl php7.2-gd php7.2-json php7.2-mbstring php7.2-mysql php7.2-snmp php7.2-xml php7.2-zip python-memcache python-mysqldb rrdtool snmp snmpd whois.

Når vi har hentet vores prerequisites opretter vi en LibreNMS user og gør den medlem af gruppen www-data:

useradd librenms -d /opt/librenms -M -r

usermod -a -G librenms www-data

Derefter installere vi Libre nms i opt mappen:

cd /opt

composer create-project --no-dev --keep-vcs librenms/librenms librenms dev-master

Så konfigurere vi databasen. I vores tilfælde bruger vi MariaDB. Vi starter med at logge ind i databasen med root:

mysql -uroot -p

Derefter laver vi databasen librenms, laver et username og password, giver fuld rettigheder til brugeren og slutter med at flushe/opdatere informationerne i databasen:

CREATE DATABASE librenms CHARACTER SET utf8 COLLATE utf8\_unicode\_ci;  
CREATE USER 'administrator'@'localhost' IDENTIFIED BY 'Password1';  
GRANT ALL PRIVILEGES ON administrator.\* TO 'librenms'@'localhost';  
FLUSH PRIVILEGES;

Derefter åbner vi tekst filen:

nano /etc/mysql/mariadb.conf.d/50-server.cnf

Vi indsætter følgende:

innodb\_file\_per\_table=1  
lower\_case\_table\_names=0

Genstarter databasen:

systemctl restart mysql

Derefter kan vi konfigurere web serveren som er vores user interface. VI bruger PHP.

Vi starter med at sikre os at tidszonen står rigtigt da det vil give os en fejl senere i validation sektionen:

timedatectl set-timezone Europe/Copenhagen.

Derefter sørger vi for at vores mods køre fra de prerequisites vi installeret:

a2enmod php7.2  
a2dismod mpm\_event  
a2enmod mpm\_prefork

Nu kan vi konfigurere apache. Vi editere tekst filen:

Nano /etc/apache2/sites-available/librenms.conf

Tilføjer apache konfigurationen som peger på serveren:

<VirtualHost \*:80>  
 DocumentRoot /opt/librenms/html/  
 ServerName 192.168.100.40  
  
 AllowEncodedSlashes NoDecode  
 <Directory "/opt/librenms/html/">  
 Require all granted  
 AllowOverride All  
 Options FollowSymLinks MultiViews  
 </Directory>  
</VirtualHost>

I LibreNMS findes der en default tekstfil som skal fjernes ellers vil den nye apache konfiguration ikke træde i kraft. Dette problem døjede med længe indtil jeg fandt ud af at den skulle erstattes via følgende command:

a2dissite 000-default

Derefter gemmer vi og genstarter apache:

a2ensite librenms.conf  
a2enmod rewrite  
systemctl restart apache2

Nu mangler vi kun at konfigurere SNMP. Først kopiere vi den nuværende konfiguration og laver den om til /etc/snmp/snmpd.conf.

cp /opt/librenms/snmpd.conf.example /etc/snmp/snmpd.conf

Derefter åbner vi tekstfilen:

nano /etc/snmp/snmpd.conf

Her kan vi sætte vores community string som vi sætter til “FJRead”. Dette er for at kunne monitor localhost.

Efterfølgende downloader vi SNMP master agenten som finder ud af hvilken styresystem det er vi prøver at trække information ud af.

curl -o /usr/bin/distro <https://raw.githubusercontent.com/librenms/librenms-agent/master/snmp/distro>

Vi tilføjer rettigheder og genstarter snmpd.

chmod +x /usr/bin/distro  
systemctl restart snmpd

Vi tilføjer default cron job, som er det polling interval vi bruge til at trække information ud af vores devices.

cp /opt/librenms/librenms.nonroot.cron /etc/cron.d/librenms

Nu kan vi komme ind på vores NMS server ved at tilgå den via:

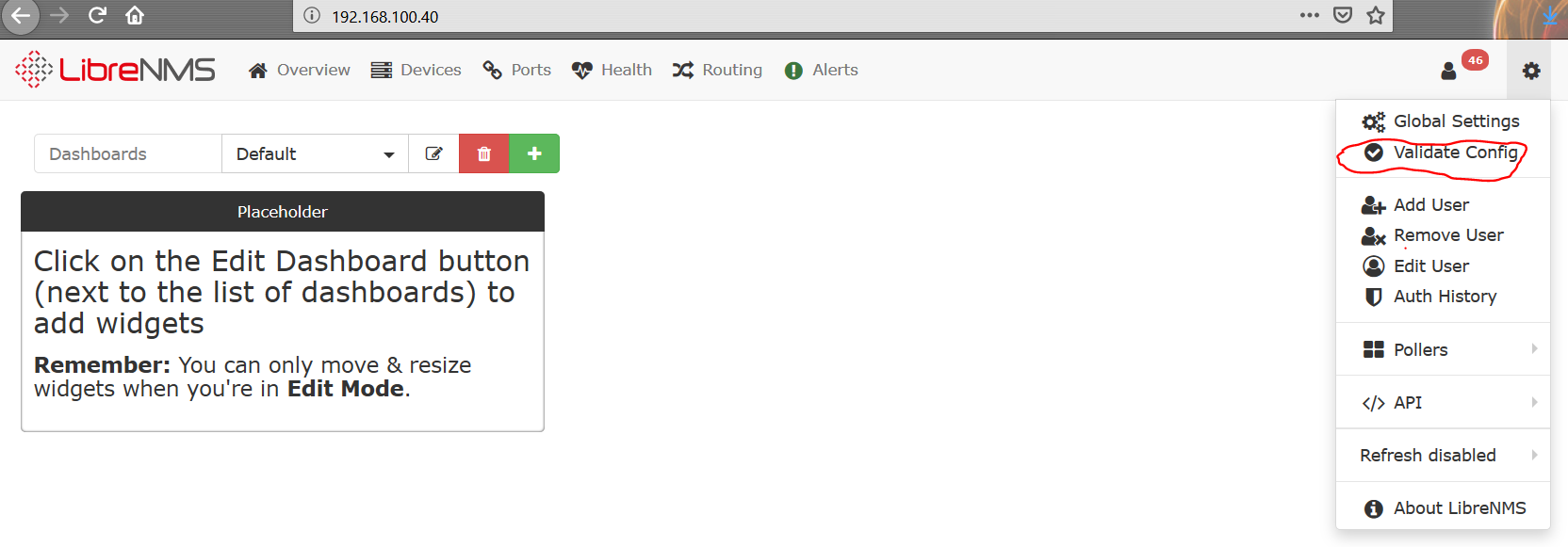
<http://192.168.100.40/install.php>

Vi kører igennem basic opsætningen og laver en ny login account. Til sidst vil den brokke sig over at der ikke er lavet en config.php file med den pågældende kode som bliver vist på skærmen. Derfor går man ind i Librenms mappen og opretter en fil der hedder:

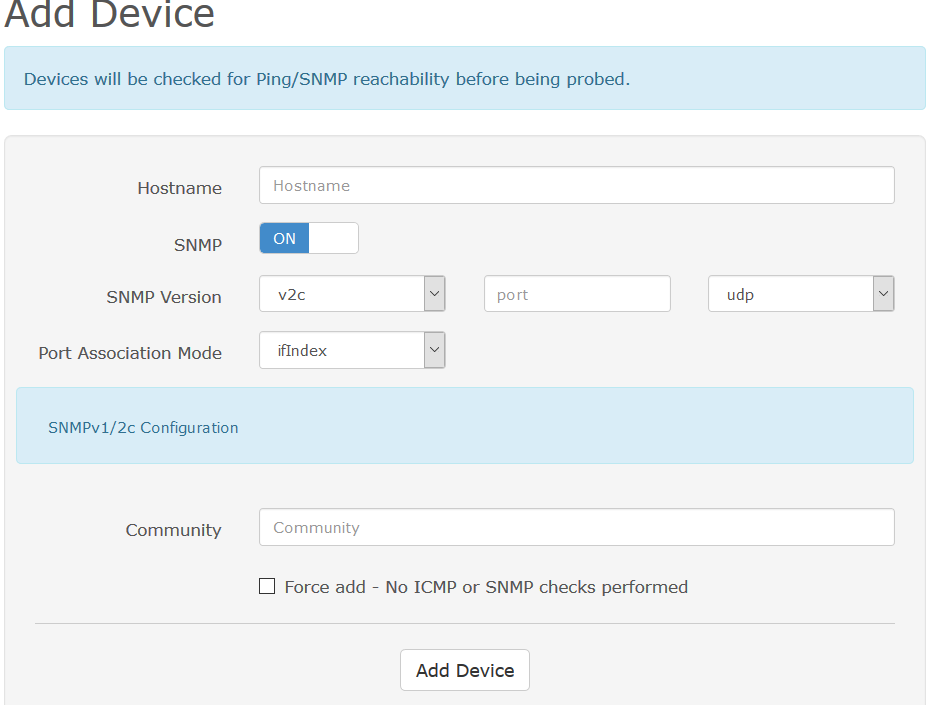
config.php

Derefter åbner man tekstfilen og indsætter koden som vist i vinduet og gemmer det.

Nu er serveren lavet og man lige validere at man har gjort alting rigtigt ved at klikke oppe i højre hjørne på validate config.



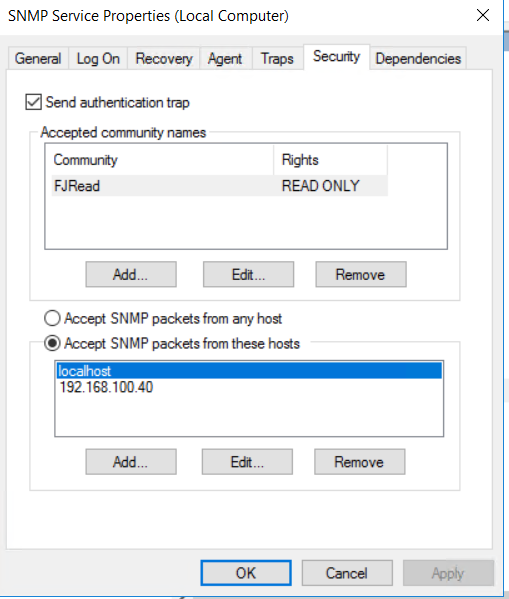
Derefter kan man oprette sine devices ved at trykke add new device og tilføje sit device via sin community string og ip adresse og derefter vil NMS starte med at poll fra device, men for at kunne adde devices er man nød til, at sætte community strings op på sine devices.



Opsætning af community strings:

Windows:

For at sætte community string op på Windows går. Installere vi servicen SNMP service. Derefter kan vi åbne services og finde SNMP. Vi tilføjer vores NMS server så den kun kan modtage SNMP pakker fra denne enhed. Derefter tilføjer vi Read only strengen “FJRead”



Cisco:

For at opsætte community strings på vores cisco enheder bruger vi kommandoen:

#snmp-server community FJRead RO.

Linux:

Først installere vi snmp:

apt-get install snmpd

Derefter sletter vi configurationen fra default filen:

Echo > /etc/snmp/snmpd.conf

Så opretter vi vores egen konfiguration:

nano /etc/snmp/snmpd.conf

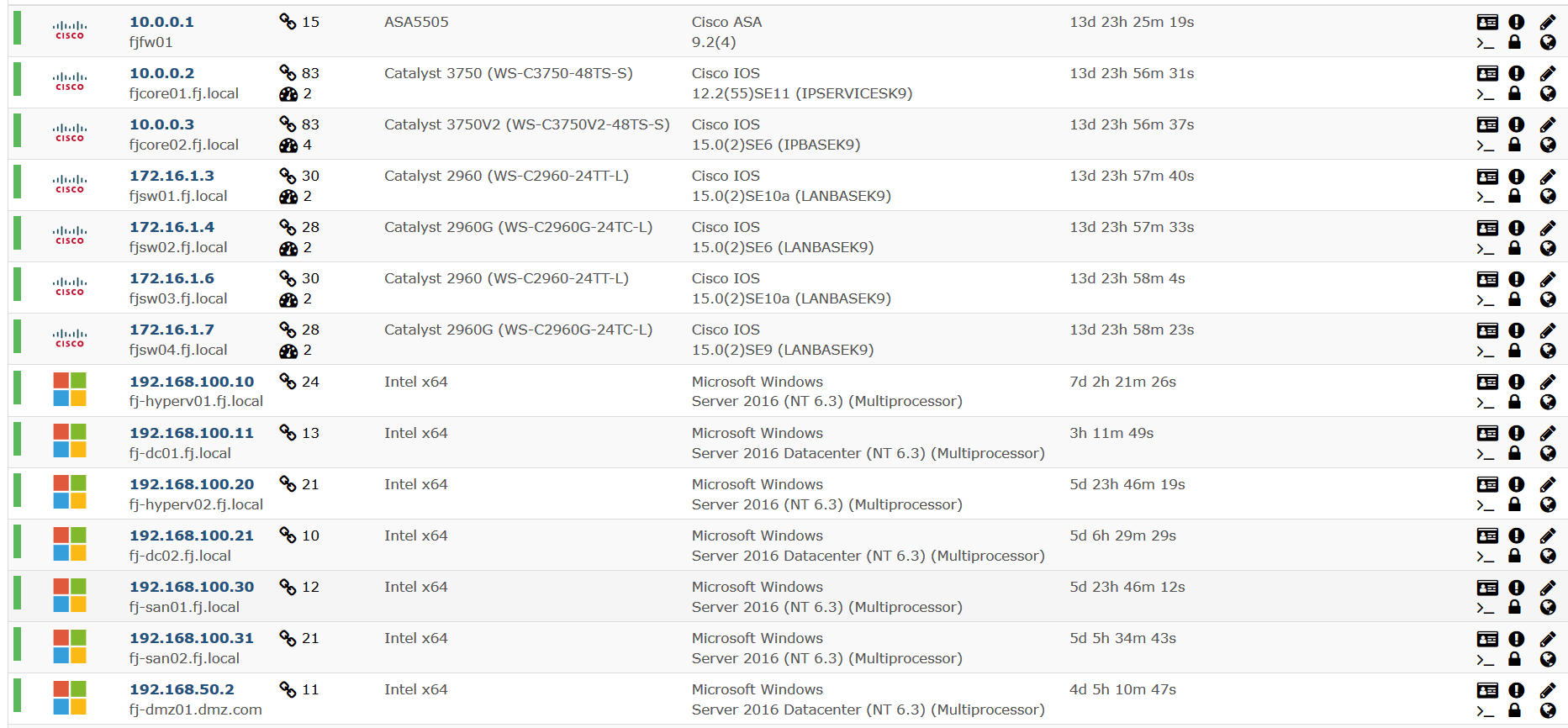
Derefter tilføjer vi community string, lokationen og kontakt serveren:

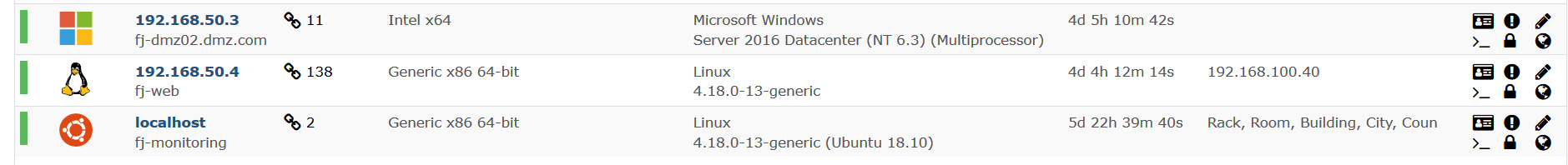
rocommunity FJRead

sysLokation 192.168.100.40

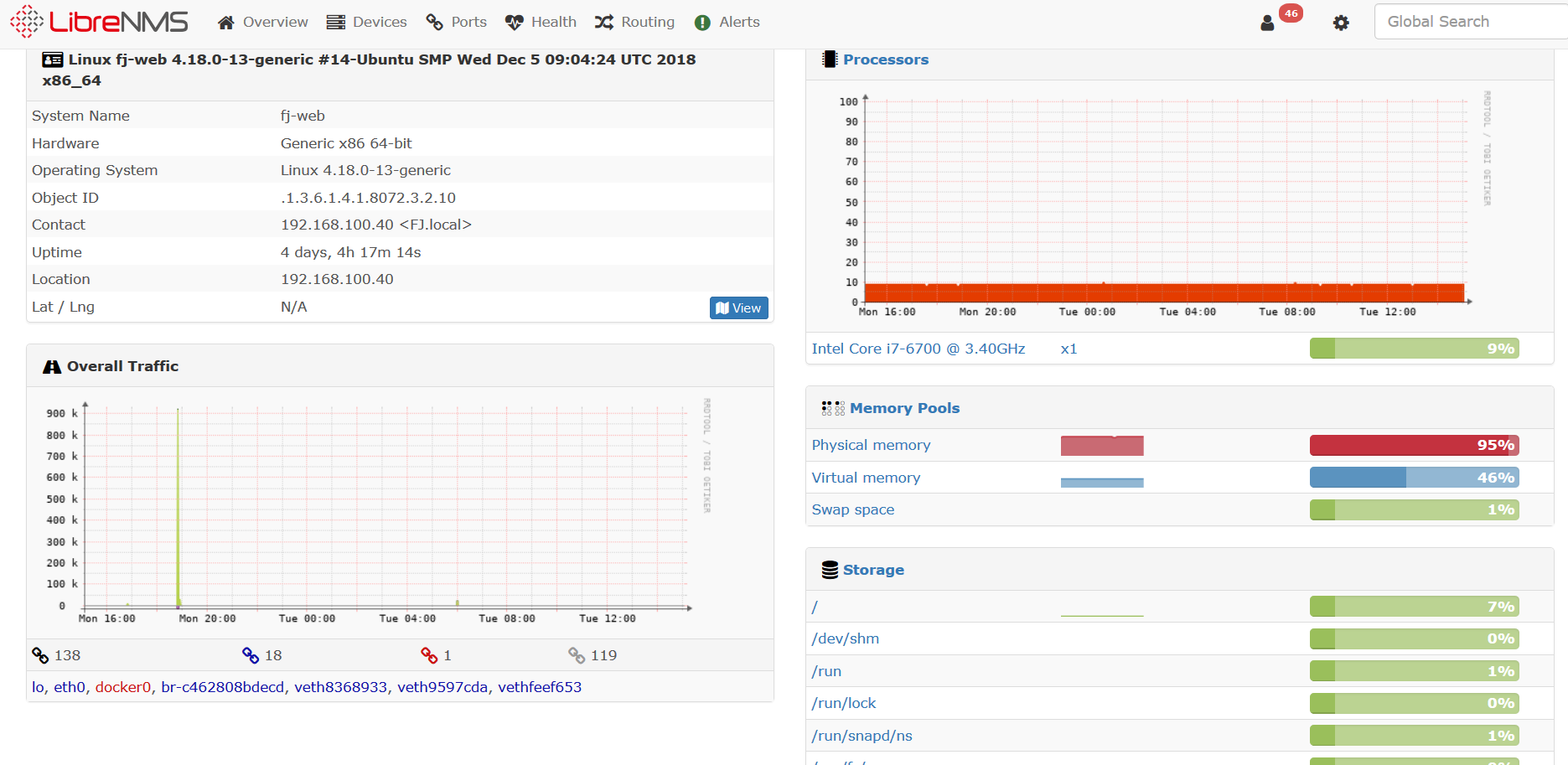
sysContact 192.168.100.40

Efterfølgende tilføjer vi alle vores enheder i vores NMS





Vi kan derefter overvåge vores servere som fx her på FJ-web.



Vi kan blandt andet se, hvor meget storage vi har og hvor meget hukommelse vi bruger.