Todo list

Remove List Todo Package Before Submission

DBD

Andreas Zoega Vesterborg Vikke
cph-av105

(Asger Hermind Sørensen)
cph-as466

(Martin Eli Frederiksen)
cph-mf237

(William Sehested Huusfeldt)
cph-wh106

Juni 2021

1 Indledning

Indhold

2 How To Run

Applikationen er lavet som en docker-compose fil for at gøre det nemmere og mere robust at køre. Det første der skal gøres er at sætte hvor meget RAM og CPU containerne må bruge. Dette gøres ved at åbne "docker-compose.yml" og sætte "mem_limit" og "cpus" under "x-shared-limit". Disse tal bliver sat én gang men bliver brugt til alle 12 container som bliver opsat.

mem_lmit = 512m
cpus = 0.5
---
512 * 12 / 1024 = 6gb memory
0.5 * 12 = 6 CPU

Efter at have sat limits kan man starte applikationen med denne kommando:

docker-compose up
'Vigtig at have bindestreg mellem docker og compose, for at gøre brug af de satte limits.'

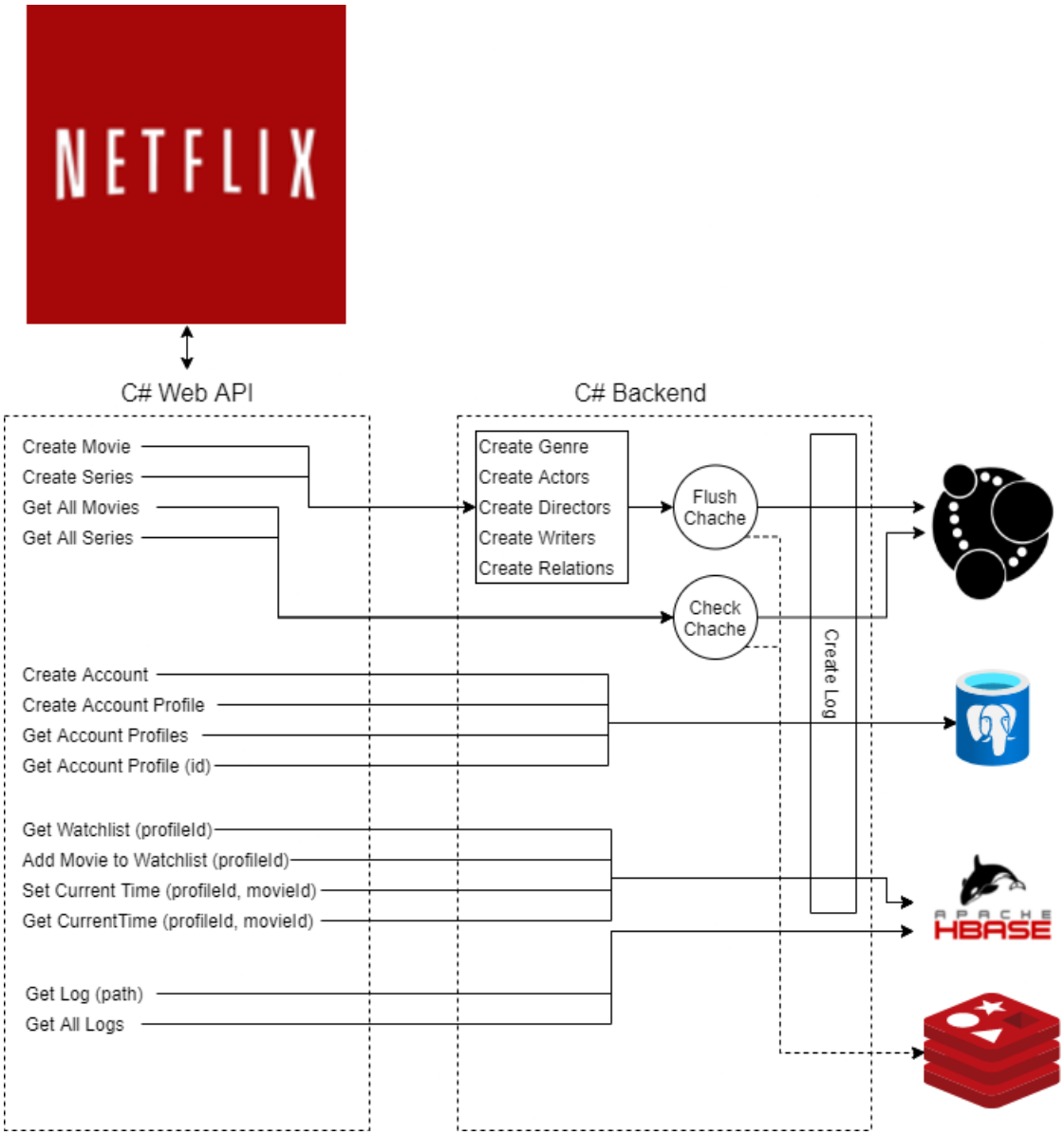
Selve applikationen vil tage noget tid at køre (omkring 5-10 minutter afhængig af hvor mange ressourcer du har allokeret i step 1). Beskeden "All Databases is up and running. . . ." betyder at alle container køre og at REST API'et har forbindelse til dem. Herfra kan du åbne din browser og navigere til: http://localhost:8000/swagger Hvor der vil blive fremvist et Swagger API med alle de kald der er opsat. I næste afsnit REST API kan du se hvor de forskellige endpoints føre dig hen og hvilke databaser de snakker med.

3 REST API

Vores API består af 3 dele;

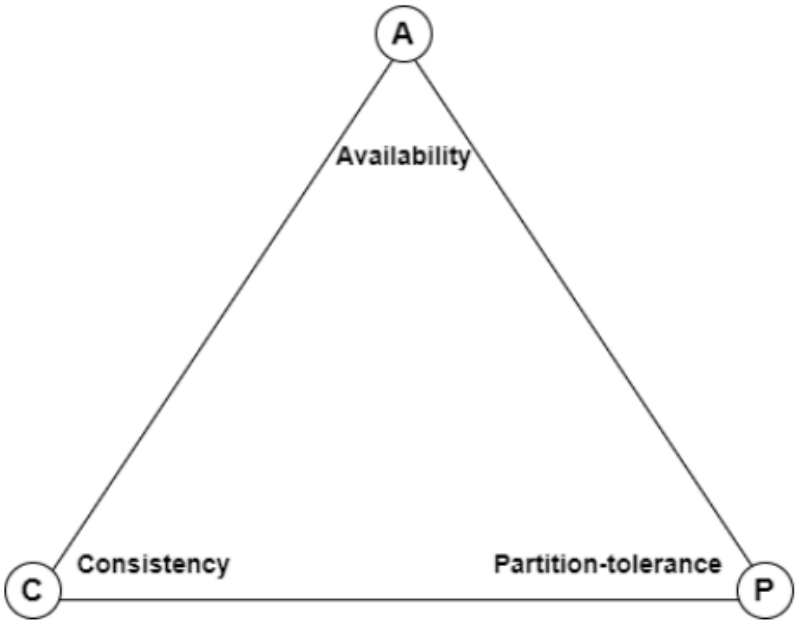
- Controllers der opsætter REST endpoints og konvertere data fra URLencoded / JSON til C# objekter.
- Services der connecter til de forskellige databaser, og om skriver dataen til queries.
- Et database lag, bestående af 4 databaser som er opsat i cluster hvis muligt.

På figur 1 har vi REST API'et med alle controllerens endpoints helt til venstre, efterfulgt af alle services' som håndtere dataen og sender det til den rigtige database. Alle endpoints, udover logs, går igennem "Create Log" servicen som skriver til HBASE. Udover dette har vi også 2 cache services' som flusher og checker om en cache findes inden den går til databasen for at finde dataen.



Figur 1: REST API Model

4 CAP Theorem



Figur 2: CAP Theorem

CAP teorien er en database teori lavet af Eric Brewer i 1999, som bygger på at en database kan vælge mellem to af de tre punkter, som illustreret i figur XX. De tre punkter står for:

- C: Konsistent: Hver læsning modtager altid den seneste skrivning eller en fejl.
- A: Tilgængelighed: Hvert request modtager et validt svar, uden garanti for at det er den seneste skrivning.
- P: Partitionstolerance: Systemet fortsætter med at virke uanset om et vilkårligt antal af beskeder bliver forsinket eller forsvinder på netværket imellem noderne.

Forskellige databasetyper går på kompromi med en af disse garantier. En database kan derfor enten være AC, AP eller CP. Hver har sine styrker og svagheder, som man skal tage i betragtning når man laver database design. CAP teorien bliver ofte refereret til sammen med ACID og BASE, som er databasemodeller der fundamentalt betegner hvordan en database håndtere begrænsningerne af CAP. Som eksempel på dette så er SQL databaser og de fleste NoSQL Graph databaser benytter sig af AC, som gør dem ACID kompatible, og dermed gør at de kan levere konsistens og tilgængelighed mellem noder i databasen. Dette er ikke muligt hvis der er partitioner, og gør at der ikke kan sikres fejltolerance som man får fra partitionstolerance.

5 Neo4j

CAP: *CA - Konsistent med en høj tilgængelighed (ACID kompatibel)*

Dette er en NoSQL databasetype, mere specifikt er det en graph database. Sammen med andre NoSQL databaser kan denne skaleres horisontalt og har en ustruktureret datastruktur, der blandt andet gør det muligt at lagre forskellige formater af data. Graph databaser er pr. design lavet til data med store mængder af relationer, hvorfor det var oplagt at benytte sig af denne database type til lagring af information og relationer mellem film, serier, skuespillere, forfattere osv. Eftersom al dette data deler mange relationer indbyrdes, eksempelvis har skuespillere relationer til de film og tv-serier de medvirker i. En anden fordel ved

denne database type er dens fleksibilitet vedr. dataformatet, hvilket på sigt kan ændre sig alt efter behov. Neo4J og graph databaser bruges dog ikke til at lagre selve mediet men man kan derimod have en attribut på en node i Neo4J der peger mod medie-filen.

Til databasen er der udviklet to datamodeller: en for film og en for tv-serier. Der er ikke udviklet datamodeller til skuespillere, instruktører eller manuskriptforfattere på grund af, at det er relationen mellem dem og filmen/serien der er vigtig, og der ikke er fokus på dybere information på de medvirkende. De to datamodeller ses her:

```
1         public class MovieModel
2         {
3             public string Title { get; set; }
4             public string ReleaseYear { get; set; }
5             public string Description { get; set; }
6             public string genre { get; set; }
7             public List<string> actors {get; set;}
8             public List<string> directors {get; set;}
9             public List<string> writers {get; set;}
10        }
11
```

Listing 1: Logs HBASE Model

```
1         public class SeriesModel
2         {
3             public string Title { get; set; }
4             public string ReleaseYear { get; set; }
5             public string Description { get; set; }
6             public string genre { get; set; }
7             public List<string> actors {get; set;}
8             public List<string> directors {get; set;}
9             public List<string> writers {get; set;}
10            public int seasons {get; set;}
11        }
12
```

Listing 2: Logs HBASE Model

De to modeller ligner til forveksling hinanden meget, men tv-serie modellen afviger fra MovieModel ved at have en 'seasons' attribut, der viser hvor mange sæsoner den respektive tv-serie har. Hver tv-serie og film node i databasen består af attributterne title, ReleaseYear, Description og Genre hvor tv-serier også har Seasons. For actors, directors og writers, så er disse ikke attributer direkte på noden, men derimod en node for sig selv hvor den specifikke tv-serie eller film har en relation til. Fordelen med at tage brug af Neo4J her er, at man let kan lave søgninger på bestemte typer af relations, således kan man hurtigt få information på hvilke noder der har f.eks. en "DIRECTED_BY" relation til en film. Det gør søgning på f.eks. alle film indenfor en genre hurtig og intuitiv.

6 PostgreSQL

CAP: *CA - Konsistent med en høj tilgængelighed (ACID kompatibel)*

PostgreSQL som også bliver kaldt Postgres er et open-source objekt-relationelt databasesystem som blev startet på University of California, Berkeley. Postgres er kendt for deres pålidelighed og dataintegritet som gør det et ekseptionelt valg til at opbevare kundedata. Postgres har udover deres mange features også været ACID-kompatibel siden 2001, hvilken er vigtigt når der er at gøre med kundedata. Postgres skriver ydermere også på deres hjemmeside [1].

> PostgreSQL has been proven to be highly scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate.

På Netflix er der per account også x antal profiler, som har hver deres mængde data. Vi har valgt at gemme disse profiler sammen med accounten i Postgres, hvor vi ved at lave en relation kan hente alle profiler for en specifik account idet man for eksempel logger ind.

7 HBASE

CAP: *CP - Konsistent med en høj partitions tolerance*

HBase er en kolonne orienteret database som er bygget ovenpå Hadoops File Systems(HDFS). Det er et open-source projekt og da der er tale om en nosql database er det muligt at skalere databasen horisontalt. HBases datamodel ligner Google's "big table" som er designet til at give en hurtig adgang til store mængder struktureret data og samtidig udnytter HBase HDFS' fejltolerance. Desuden er HBase en del af Hadoops ecosystem som udbyder real-time adgang til læsning og skrivning af data til HDFS. Det er muligt at gemme data direkte i HDFS eller gennem HBase men for at kunne læse data fra HDFS skal det gå igennem HBase. I vores projekt har vi valgt at bruge HBase til at gemme den data der beskriver hvor langt en bruger er kommet med en given film eller serie. Nedenfor ses database modellen:

```
watchlist
    <profile id>
        <movie/series id>:type = <type>
        <movie/series id>:timestamp = <timestamp>
        <movie/series id>:season_id = <season id>
        <movie/series id>:episode_id = <episode id>

watchlist
    1234
        5:type = movie
        5:timestamp = 1600
        2:type = series
        2:timestamp = 1200
        2:season_id = 3
        2:episode_id = 2
    1235
        5:type = movie
        5:timestamp = 45000
        4:type = movie
        4:timestamp = 200
```

Listing 3: Watchlist HBASE Model

På figur XXX ser vi at vi har en en overordnet tabel kaldet watchlist i denne tabel gemmer vi så profil id'et som vores række nøgle efterfulgt af id'et på den pågældende film eller serie. Derudover gemmer vi også hvilken type materiale der er tale om altså film eller serie samt et timestamp på hvor lang denne profil er kommet med den givne film eller serie. Hvis der er tale om en serie gemmer vi samtidig id'et på afsnittet der er tale om og id'et på hvilken sæson vi finder afsnittet i. På figur XXX ses også et eksempel på en profil med id 1234 hvor det fremgår at denne profil har set en film med id 5 og har timestampet 1600. Det fremgår også at samme profil er i gang med at se en serie der har id 2 hvor der er tale om det 2. afsnit i den 3. sæson og afsnittet har så et timestamp på 1200. På figur XXX ses ydermere endnu en profil der er i gang med at se to film med hhv. id 4 og 5.

Udover at gemme en "liste" i HBase over hvor langt hver profil er kommet med diverse film og serier gemmer vi også en "liste" af logs på api kald.

```
log
    <http_method>
        <path>:<epoc_timestamp> = <data>

log
    GET
        /api/Account/get:1621847222436 = {Id: 1}
        /api/Account/get:1621847222435 = {Id: 5}
        /api/Account/get/all:1621847222436 = {}
    POST
        /api/Account/create:1621847222478 = {JSON DATA}
        /api/Account/create:1621847222578 = {JSON DATA}
        /api/Account/create:1621847222443 = {JSON DATA}
```

Listing 4: Logs HBASE Model

På figur XXX ser vi en tabel med navnet log som har en http metode som række nøgle efterfulgt af vejen der er brugt til http metoden som familie id samt et timestamp skrevet som unix timestamp med tilhørende json data. På figur XXX fremgår der også to eksempler af tabellen log hvor der i det første eksempel er blevet kaldet en get metode på /api/Account/get med et tilhørende unix timestamp og json data(i dette tilfælde er json dataen det id på den film eller serie brugeren ønsker at se). Det andet eksempel er en en post metode som skal symbolisere en brugeroprettelse hvor json dataen indeholder de nødvendige oplysninger for at oprette en bruger.

8 Redis

CAP: *CP - Konsistent med en høj partitions tolerance*

```
cache:<cache_type>#<genre ifexists> = <json data>

cache:Movie = {JSON DATA}
cache:MovieByGenre#Fantasy = {JSON DATA}
cache:Series = {JSON DATA}
cache:SeriesByGenre#Fantasy = {JSON DATA}
```

Listing 5: Logs HBASE Model

Litteratur

[1] PostgreSQL. What is postgresql? https://www.postgresql.org/about/, 2021. [Online; accessed 30-May-2021].