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.'

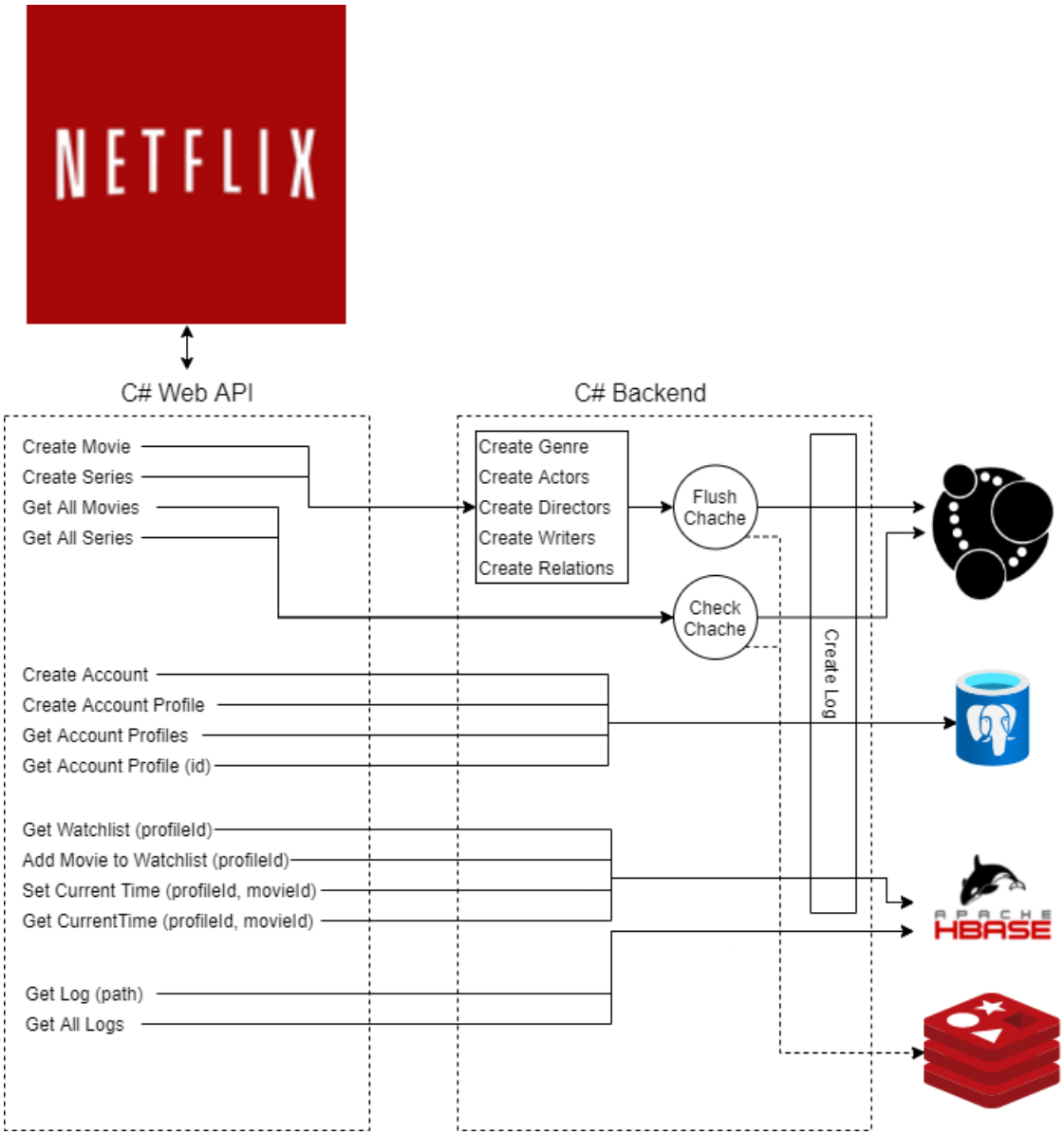
Selve applikationen vil tage noget tid at køre (omkring 5-10 minutter afhængig af hvor mange ressourcer du har allokeret i step 1). Beskeden "All Databases is up and running. . . ." betyder at alle container køre og at REST API'et har forbindelse til dem. Herfra kan du åbne din browser og navigere til: 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;

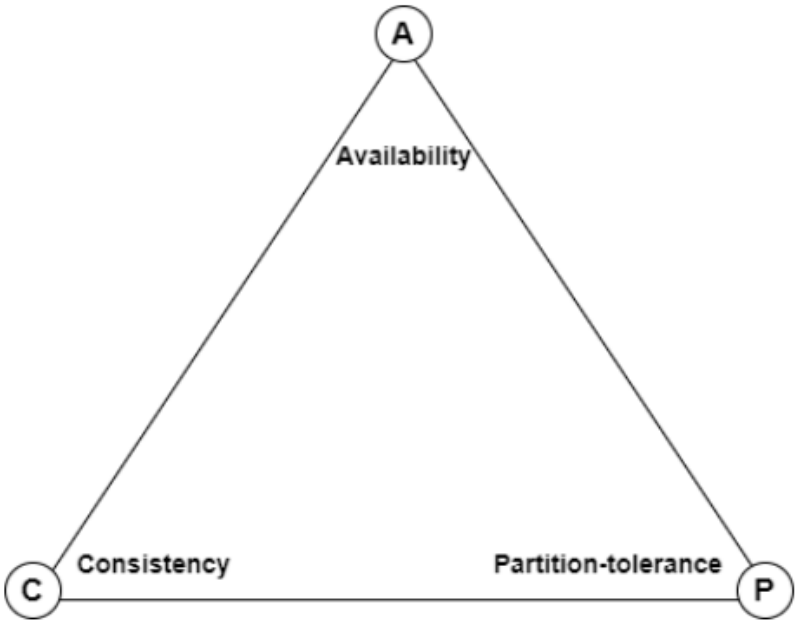
- Controllers der opsætter REST endpoints og konvertere data fra URLencoded / JSON til C# objekter.
- Services der connecter til de forskellige databaser, og om skriver dataen til queries.
- Et database lag, bestående af 4 databaser som er opsat i cluster hvis muligt.

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 teorien



Figur 2: CAP teorien

I snakken om databaser anses CAP-theorem som værende en af de fundamentale teorier, og er særligt brugt i beslutningsprocessen om hvilke databaser der skal tages i brug. Teorien blev præsenteret i 2000 af Eric Brewer[1] der viser, at et distribueret system kun kan opfylde to ud af tre garantier. Garantierne er illustreret i figur 2, og lyder:

- C: Konsistent: Hvert read-request modtager altid et svar med det nyeste data eller en fejlmeddelelse.
- A: Tilgængelighed: Hvert request modtager et validt svar, uden garanti for at det er det seneste data.
- P: Partitionstolerance: Systemet skal kunne fortsætte med at fungere på trods af kommunikation sammenbrud mellem noderne i systemet.

Ifølge logikken bag teorien, kan et databasesystem ikke opfylde alle tre garantier, hvilket medfører at en database må gå på kompromis med en garanti. En database kan derfor enten være AC, AP eller CP. Hver har sine fordele og ulemper, som bør tages til eftertanke. CAP teorien bliver ofte refereret til sammen med ACID og BASE, som er databasemodeller der fundamentalt betegner hvordan en database håndterer begrænsningerne af CAP. Eksempelvis vil SQL databaser og graph databaser ifølge teorien som udgangspunkt opfylde tilgængelighed og konsistent garantierne, hvilket gør dem ACID kompatible. Dermed er dataen højt tilgængelig og sikret at det er det nyeste data. Ligeledes betyder dette at partitionstolerance ikke kan opfyldes.

5 Neo4j

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

Neo4j er en NoSQL databasetype, mere specifikt er det en graph database. Sammen med andre NoSQL databaser kan denne skaleres horisontalt [3], og har en semi-struktureret 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 neo4j til lagring af information på og relationer mellem film, serier, skuespillere, forfattere osv. Eftersom al dette data deler mange relationer indbyrdes, eksempelvis

har skuespillere relationer til de film og 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 mediefilen [2].

Til databasen er der udviklet to datamodeller: en for film og en for 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. De to datamodeller ses på listing 1 og 2:

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

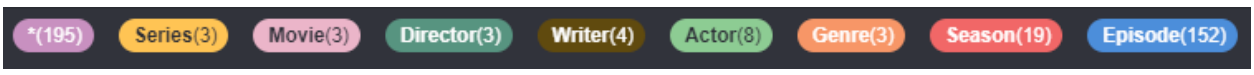
Listing 1: C# MovieModel Class

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

Listing 2: C# Seriesodel Class

De to modeller ligner til forveksling hinanden meget, men serie modellen afviger fra MovieModel ved at have en 'seasons' attribut, der viser hvor mange sæsoner den respektive serie har. Hver serie og film node i databasen består af attributterne title, ReleaseYear, Description og Genre hvor serier også har Seasons. For actors, directors, writers og genre, så er disse ikke attributter direkte på noden, men derimod en node for sig selv hvor den specifikke serie eller film har en relation til.

Struktur af noderne



Figur 3: Farvekoder til neo4j.

For at gøre det nemmere at gennemskue den data der ligger i neo4j, kan noderne inddeles efter en farvekode og størrelse. Mere relevante og overordnede noder som genre, film og serier bliver større, mens underordnede noder som episoder bliver små.

Figur 4: Film og serie tilknyttet til "Superhero" genren.

På figur 4 ses et udsnit af neo4j databasen, med en enkelt film og serie samt deres tilhørende børne-noder, som er inddelt efter farve som set på figur 3. Film og serier der bliver oprettet bliver knyttet til en genre, og derefter knyttes medvirkende til en film/serie. En serie har derudover relationer til sæsoner, og de sæsoner en relation til episoder, jo længere væk fra forældre-noden en node er, jo mindre bliver den. Fordelen ved neo4j er, at man let kan lave søgninger på disse relationer, og således hurtigt kan få informationer af interesse frem. Dermed gør det dét muligt at kunne søge på film og serier efter genre, skuespiller mm. på en hurtig og intuitiv måde.

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. [4] 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 [4].

> 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 hver har deres mængde forskellig data. Vi har valgt at gemme disse profiler sammen med accounten i Postgres, så vi ved at lave en relation kan hente alle profiler for en specifik account. På listing 3 og 4 nedenfor kan SQL de to create scripts ses. Account har en primary key som bliver automatisk inkrementeret, som vi i profiles kan bruge til at lave en foreign key. På denne måde vil hver profil være knyttet til en account.

```
1         CREATE TABLE IF NOT EXISTS accounts (
2             account_id INT GENERATED ALWAYS AS IDENTITY,
3             email varchar(255) UNIQUE NOT NULL,
4             password varchar(200) NOT NULL,
5             firstname varchar(50) NOT NULL,
6             lastname varchar(50) NOT NULL,
7             PRIMARY KEY(account_id)
8         );
9
```

Listing 3: Logs HBASE Model

```
1         CREATE TABLE IF NOT EXISTS profiles (
2             profile_id INT GENERATED ALWAYS AS IDENTITY,
3             account_id INT NOT NULL,
4             name varchar(50) NOT NULL,
5             age int NOT NULL,
6             PRIMARY KEY(profile_id),
7             CONSTRAINT fk_account
8                 FOREIGN KEY(account_id)
9                     REFERENCES accounts(account_id)
10        );
11
```

Listing 4: Logs HBASE Model

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 5: 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 6: 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 7: Logs HBASE Model

Litteratur

[1] IBM. Cap theorem. https://www.ibm.com/cloud/learn/cap-theorem, 2021. [Online; accessed 31-May-2021].

[2] Neo4j Staff. Storing media files in neo4j? https://community.neo4j.com/t/storing-media-files-in-neo4j/10475, 2019. [Online; accessed 31-May-2021].

[3] Paul Gillin. Neo4j gives its graph database a major enterprise facelift. https://siliconangle.com/2020/02/04/neo4j-gives-graph-database-major-enterprise-facelift/, 2020. [Online; accessed 31-May-2021].

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