

SOFTWARETEKNOLOGI

4. SEMESTERPROJEKT

PRJ4 - Samlet

Author:

Oliver SPANGE - au675896
Anders FRIIS - au674304 -
202006702
Patrick LARSEN - au649748 -
201903892
Mads Kristian Key Høgh
CHRISTENSEN - au669782 -
202007524
Rasmus AAGAARD - au532459 -
201510642
Casper Bjerregaard SAMBLEBEN -
au664751 - 202004917
Andreas Steen Simmelkiær
SØRENSEN - au196631

Supervisor:
Lars Mortensen

Character Count: 70046 characters with spaces
24. oktober 2022

Resumé

Det følgende dokument er udarbejdet af projektgruppe 6, i forbindelse med Semesterprojekt 4(SW4PRJ4) for softwareteknologistuderende. Til udførelsen af projektet er der brugt metoder og teknikker lært igennem det nuværende og de tidligere semestre. Igennem projektet er der arbejdet med forskellige arbejdsmetoder, projektorganisering og arbejdsfordeling. Dette er beskrevet igennem rapporten. Formålet med projektet har været at udvikle en hjemmeside med formålet at kunne assistere en bruger med at finde et forslag til en ny film ud fra valgte film. Ydermere skal produktet virke som et bibliotek, hvor en bruger kan søge efter film og finde informationer om den. Systemet består af en SQL database der indeholder filmene og den data der hører til dem, en frontend lavet i razor pages, og en backend der består af DiscoverAlgoritmen der finder filmforslag.

Systemet er blevet testet ved brug af flere metoder. Der er brugt unit tests som en del af continuous integration, der er blevet lavet en række modultests for de forskellige elementer, og til sidst er der lavet en integrationstest, der leder op til udførelsen af accepttestene, som alle er bestået.

Abstract

The following document has been prepared by project group 6, in connection with Semester Project 4 (SW4PRJ4) for software technology students. For the execution of the project, methods and techniques learned throughout the current and previous semesters have been used. Throughout the project, work has been done using different methods, project organization and division of labor. This is all described throughout the report and the included appendix. The purpose of the project has been to develop a website with the purpose of being able to assist a user in finding a proposal for a new movie based on a selection of films. Furthermore, it should act as a catalog where a user can search for movies and find information about it. The system consists of a SQL database that contains the movies and the data that belongs to them, a frontend made in razor pages, and a backend that consists of the discover algorithm that finds the movie suggestions.

The system has been tested using several methods. Unit tests have been used as part of continuous integration, a number of module tests have been made for the various elements, and finally an integration test has been made that leads up to the execution of the acceptance tests, all of which have been passed.

Contents

1	Indledning	1
2	Problemformulering	1
2.1	Problemstilling	1
2.2	Uddybelse	1
3	User Stories	2
3.1	Første iteration	2
3.2	Anden iterationer - Summary user stories	2
4	Kravspecifikation	4
5	Risiko Analyse	5
5.1	Database	5
5.2	Applikationstype til FrontEnd	5
5.3	Ekstern API	6
6	Projektbeskrivelse	7
6.1	Gruppeddannelse	7
6.2	Process	7
6.2.1	Scrum	7
6.2.2	Redmine	8
6.2.3	Git	8
7	Arkitektur	10
7.1	Systemarkitektur	10
7.2	Softwarearkitektur	11
7.2.1	System Context niveau	11
7.2.2	Container niveau	12
7.2.3	Component niveau	13
7.2.4	Code niveau	15
8	Analyse af Software	17
8.1	Discover Algoritme	17
8.1.1	Filtre	17
8.2	Search Bar	17
8.3	Moviepage	19
8.4	Database Struktur	19
8.5	Acquire Application	20
8.6	NUnit	21
9	Design og implementering	22
9.1	Indledning	22
9.2	Catalogue	22
9.2.1	Implementering Frontend	22
9.2.2	Implementering Backend	22
9.3	Discover	23
9.3.1	Implementering Frontend	23
9.3.2	Implementering Backend	24
9.4	AutoComplete	26
9.4.1	Implementering af AutoComplete frontend script	26
9.4.2	Implementering af AutoComplete backend controller	27
9.5	Databasen	28

9.5.1	Implementering af Databasen	28
9.6	Design af NUnit testsuite	29
10	Test af software	31
10.1	Unittest af Discover Algorithm	31
10.2	Unittest af Search	32
10.3	Modultest	32
10.3.1	Search bar	32
10.3.2	Movie page	37
10.3.3	Discover	38
10.4	Acquire	41
10.5	Integrationstest	41
10.6	Accepttest	42
11	Perspektivering	43
11.1	Systemarkitektur	43
11.2	Discover-Algoritmen	43
11.3	Continuous Integration	44
11.4	Statisk database	44
11.5	FrontEnd	44
12	Konklusion	46
13	Personlige Konklusioner	47
13.1	Oliver	47
13.2	Anders	47
13.3	Patrick	47
13.4	Mads	47
13.5	Rasmus	47
13.6	Casper	48
13.7	Andreas	48
A	Bilag	49

Ordforklaringsliste

Term	Forklaring
DiscoverMovies	Navnt for det samlede betegnelse for det udviklede system.
Catalogue Page	Forsiden på webapplikationen/Indexpage, der viser et katalog over databasens film.
Movie Page	Side der præsenterer relevant data for en specifik film.
Discover Page	Side, med funktionalitet til at give filmforslag baseret på 5 valgte film.
DiscoverAlgorithm	Algoritme med specialiserede filtre der danner forslaget til Discover Page.
TMDB	The movie database, Database med film det bliver hentet til projektet.
MVC	Model-View-Controller design pattern
SPA	Single Page application design pattern
SRP	Single Responsibility Principle design pattern
Searchbar	Søgebar, søgefelt og dropdown menuer på catalogue page
API	Application Programming Interface

Underskrifter

Aver Spang

Patrick Larsen

mads christensen

Casper Sandøen

Annes

Dastrik

Lars Steen Søder

Ansvarsområder

Alle de afsnit i rapporten der ikke har en tilhørende række i Tabel 1 er lavet i samarbejde af hele projektgruppen.

Ansvarsområder	Ansvarshavende
Implementering af CataloguePage + Search.	Primære: Patrick. Sekundær: Andreas
DiscoverAlgorithm	Mads, Rasmus og Andreas
SQL databasen	Primære: Patrick, Andreas og Rasmus. Sekundære: Mads, Casper, Oliver og Anders
Implementering af MoviePage	Casper og Anders
Implementering af DiscoverPage	Oliver
Implementering af Autocomplete	Andreas og Oliver
Projektbeskrivelse	Anders
Acquire Application	Andreas
NUnit af Searchbar	Patrick
NUnit af Discover	Primære: Andreas Sekundære: Oliver og Rasmus

Table 1: Oversigt over gruppemedlemmers ansvarsområder.

1 Indledning

I dagligdagen fylder underholdningsbranchen meget i det gennemsnitlige hjem. Mængden af nye udgivelser, samt et filmmarked domineret af få mastodonter^{1 2}, betyder at film med færre midler til marketing har sværere ved at få del i lampelyset. Filmlandskabet virke uoverskueligt, hvis ikke man aktivt holder overblik over nye udgivelser.

Dette projekt forsøger at løse dette problem, som bliver uddybet herunder i problemformuleringen, ved at udearbejde et produkt der kan give nogen gode filmforslag. Der har i starten af forløbet været en stærk forstilling om hvad funktionalitet produktet skulle have, dette produkt er så blevet udviklet af gruppen, igennem et forløb der har anvendt scrum som arbejdsmetode.

2 Problemformulering

2.1 Problemstilling

Streamingtjenester benytter en algoritme til anbefaling af film, som inddrager tidligere sete film samt marketingsaftaler med de større produceringsselskaber. Derudover er der en generel trend for at promovere egne film. Dette betyder at brugeren direkte indgår i en marketings strategi, der omhandler at få brugeren til at se specifikke medier, og optimere antal sete timer. Dette kan ofte lede til foreslag, der ikke giver størst mulig værdi for brugeren, men derimod størst værdi for streamingtjenesten. Foreslag ud fra et stort katalog af tidligere sete medier, tager autonomi fra brugeren hvis eksempelvis brugeren har ønske om foreslag ud fra en bestemt række film. Et tænkt eksempel kunne være en bruger der ønsker at se en gyserfilm, men tidligere har set flere romantiske komedier, som indgår i algoritmen og påvirker foreslagene brugeren vil blive præsenteret for.

Én mere direkte metode til anbefaling af film, hvor brugeren bevidst udvælger de film, som filmanbefalingerne skal bestemmes ud fra, ville kunne hjælpe brugere med at finde tilfredsstillende film til den specifikke efterspørgsel.

Et sekundært problem ved streamingtjenester er relateret til konseptet, paradox of choice³, der omhandler at den menneskelige hjerne, når præsenteret for mange muligheder er det svært at vælge én af mulighederne. Selve processen med at vælge film, kan altså effektiviseres, både således brugeren får bedre forslag ud fra selvvalgte interesser, men også så der er mindre tidsspild under valget af film. Begge disse problemer under valg af en film vil dette projekt behandle samt udvikle en prototype til en løsning af problematikkerne.

2.2 Uddybelse

Vi vil gerne lave et website, hvor en bruger kan få et forslag til en film, ud fra egen input. Dette input vil bestå af at brugeren udvælger fem film fra et katalog på websitet, som de godt kan lide, og så vil websitet give forslag på en ny film baseret på de udvalgte film.

Websitet skal altså kunne lave en søgning i databasen, på baggrund af de informationer, som kan findes på brugerens allerede udvalgte film, og bruge denne data, til at finde lignende film. Det kan f.eks. være navnene på de folk som har lavet deres yndlingsfilm, og se om en kombination af de samme folk, har lavet andre film. Når en bruger laver sådan en søgning, så vil resultaterne være begrundet, så brugeren er beviget over hvorfor lige netop de leverede resultater er blevet valgt.

Websitet ville også kunne tillade andre typer af søgninger på film, ud fra f.eks. genre, titler, hvem medvirker, el. står bag filmen mm.

For at skabe dette website, skal der både tages en database i brug, samt et .net backend program. Der skal også bruges en httpclient til at tilgå APIer, hvor yderligere information til filmene, kan hentes. Til frontenden vil der blive benyttet Razor pages til at vise både selve websitet, men også resultater fra både film søgninger, og film forslag.

¹<https://www.statista.com/statistics/187307/box-office-market-share-of-universal-in-north-america-since-2000/> [3]

²<https://www.statista.com/statistics/187300/box-office-market-share-of-disney-in-north-america-since-2000/> [2]

³<https://uxplanet.org/breaking-paradox-of-choice-netflix-case-study-7f29107d1e2b> [4]

3 User Stories

3.1 Første iteration

Feature: Find specific movie by searching

Scenario: When the user opens the webapplication through their browser.

Then the frontpage is presented, which includes a catalogue with all movies in database.

Then user searches, in the embedded search-bar, for the title, actors, director or release-year, of the specific movie.

When the movie is found the user will click on the movie

Then the movies personal page will open and present the movies data.

Feature: Find specific movie by scrolling

Scenario: When the user opens the webapplication through their browser.

Then the frontpage is presented, which includes a catalogue with all movies in database.

Then the user will scroll through the catalogue to find the specific movie.

When the movie is found the user will click on the movie

Then the movies personal page will open and present the movies data

Feature: Get Dynamic Movie Recommendation

Scenario: When the user opens the webapplication through their browser.

Then the frontpage is presented, which includes a catalogue with all movies in database.

When the user clicks "Movie Recommendation" in the navigationmenu.

Then the movie recommendation page is presented to the user.

Then the user chooses 3-5 movies through a searchbar embebed in the page. These movies are presented on the page.

Then the user clicks the "Find Recommendation"-button.

Then the webapplication recommends 1 or more movies to the user.

3.2 Anden iterationer - Summary user stories

System must have an account-page, with personal moviecollections

User navigates through the UI to the login page. The user inputs their previously defined username and password, or uses the same page to create one. When the user is logged in, they can access their own personal collection of movies.

System must included a developed API to the dynamic film recommendation

User sends api request, with the movies the user wants a suggestion for. The API then returns the suggested movie, with its relevant information.

The algorithm must have the ability to lock parameters

The search algorithm is able to lock certain parameters when searching.

The system should use an API to import posterpictures for all movies

GUI sends a request to the external API and requests the cover photo, or a url to it, this is then handled and displayed to the user.

The system should use API to import movie descriptions

GUI sends a request to the external API and requests the movie description, or a url to it, this is

then handled and displayed to the user.

The movie recommendationpage could show which parameters decides the recommendation

The reccomendation page will show which parameters the algorith used to find the results.

The system could have a "try your luck" button that presents a random movie

The user presses the random movie button, and a random movie page is shown.

System could import ratings from IMDB/Rotten Tomatoes

When the user requests a movie suggestion, the system takes the movies ratings into account when choosing the movie to suggest.

System could show which streaming services a movie is available on.

As the user selects a movie for which the user wants to see a page, a item on that page describes what streaming services the movie is available on, and anny additional information related to that. (price to rent, is the user subscribed to the service, ect.)

4 Kravspecifikation

Under udviklingen af kravspecifikationen blev der lavet en række overvejelser for at undgå feature-creep samt at lave en klar afgrænsning og dermed sikre at en proof of concept prototype blev færdiggjort.

Disse overvejelser ledte til en afgrænsning af hvilke krav der var nødvendige for at opnå ”minimal viable product”. En række af kravene er skrevet ud fra officielle krav fra projektbeskrivelsen, såsom at både DAB (Database), BED (Back-end) og FED (Front-end) kursurne skulle anvendes under udviklingsprocessen. Derudover blev der efter en risikovurdering, se afsnit Risiko Analyse, samt en vurdering af ønsket funktionalitet valgt at udvikle en web-applikation hvilket også fremgår af kravene. For at kunne løse den valgte problemstilling var det nødvendigt at have en lokal database der er repræsentativ for en årrækkes udgivede film. Derfor var det også valgt at den lokale database til produktet skulle klones fra en ekstern film-database, via en række queries. Det skrevne program med API-kaldene til den eksterne database og metoderne til lagring i den lokale database kaldes ”acquire program”.

Minimal Viable Product

- Skal indeholde database, database skal indeholde titel, folk der står bag film (instruktør, skuespillere osv.). Databasen skal indeholde film fra et bestemt årti.
- Skal indeholde GUI, der skal tilgås gennem hjemmeside.
- Skal indeholde backend, backend skal have en algoritme der laver crosscorrelation mellem valgte film og kommer med filmforslag.
- Databasen skal klones fra en ekstern database.
- GUI skal indeholde et katalog der kan søges i.
- GUI skal have en side med dynamiske filmforslag ud fra valgte film.
- Hver film skal have egen side/data.

Udviklingsmuligheder

- GUI skal indeholde et konto-system, med login med enkelte filmsamling.
- Systemet skal indeholde en api til de dynamiske filmforslag.
- Algoritme skal kunne låse visse parametre (instruktører, specifikke genrer og lign.).
- GUI burde inddrage API til coverbilleder til alle film.
- GUI burde inddrage API til filmbeskrivelser.
- Filmforslagssiden burde kunne vise hvilke parametre der lægger grundlag for forslaget.
- En random side til at finde en tilfældig side.
- Systemet kan inddrage IMDB/Rotten tomatoes rating under filmforslag.
- Systemet kan vise hvilke streamingservices, specifikke film er tilgængelige på.
- Systemet skal indeholde en side for hver skuespiller
- Filmsiderne skal kunne vise

5 Risiko Analyse

5.1 Database

Projektet med Discover movies skulle trække på data fra en ekstern database, hvor denne data skulle lægges ned i en intern database struktur. Derfor skulle der i projekts opstartsfase vælges en fornuftig database-struktur. Der er generelt mange database typer at vælge imellem, her kan f.eks. nævnes Microsoft SQL server og MongoDB. Et stort behov i projektet, var ret hurtigt at der kunne arbejdes med reelle data, hvor det samtidig var mindre vigtigt for projektet, at performance på DB'en gav hurtige dataoverførsler, ved det primært var et proof of concept. Ved at Azure Database oprettelse var blevet lukket ned fra universitets side, skulle der tænkes over hvordan dataen på DB'en blev delt gruppens medlemmer.



Figure 1: MongoDBVsMssql ⁴

Fordelene ved at vælge MongoDB med none relationel approach, at der ikke på samme vis var et fastlagt schema, og derfor kunne den interne datastruktur nemmere modificeres efter behov. Dette vil gøre designet noget mere fleksibelt og modulært. Hvis data blev svært at etablere i tabeller, ville dette være nemmere dynamisk at ændre på senere. Ved at MongoDB kan have dupligeret data, så kan dette øge performance.

En klar ulempe med MongoDB ville være, at det ikke altid er nemt at gennemskue hvor specifik data er gemt, da MongoDB er mere dynamisk i dens struktur, da der ikke er et klart fastlagt schema. Desuden var undervisningen tilrettelagt sådan at undervisning i MongoDB ville lægge ret sent på semestret i faget DAB. Hvilket var en stor problematik.

Fordelene ved at vælge MS SQL eller relationel, var at der hurtigt på semestret ville komme den nødvendige viden til at få en MS SQL Database etableret, samtidig med undervisning i benyttelse af frameworket EFcore. EFcore gav viden i etablering og håndtering af MSSQL SQL serve i C#. Desuden er MS SQL lidt mere logisk i dens udtryk, da der er et fastlagt schema, samtidig med at data til DiscoverMovies, kunne struktureres på en fornuftigt måde med et fastlagt schema.

Ulemperne ville dog være at schemaet er mere fastlagt, og derfor kunne give mindre fleksibilitet.

DiscoverMovies blev etableret med MS SQL, da en nøglefaktor for projektets database var, at der hurtigt i semestret blev etableret en velfungerende DB, da andre moduler i systemet afhæng af denne struktur med data. Desuden var undervisningen tilrettelagt så, at der kom undervisning i netop MS SQL DB struktur med EFcore først. Dette var endnu et vigtigt element i valget.

Vi fik stillet en Azure DB til rådighed lidt senere på semesteret, og derfor valgte vi denne online løsning, samtidig med at en rød tråd imellem Microsoft produkter blev overholdt.

5.2 Applikationstype til FrontEnd

DiscoverMovies er blevet udviklet med .Net, WPF blev tidligt valgt fra som mulighed, da der både var tanker om, at hoste produktet online, samt muligvis at udbygge en API, således at andre kunne til at få filmforslag, indbygget i deres hjemmeside. Dette valg har også den indbyggede fordel, at det er en del nemmere at distribuere linket til en hjemmeside, end en installer til en lokal applikation,

⁴<https://blog.sqlauthority.com/2020/04/18/sql-terms-vs-mongodb-terms/> [1]

et applikation som også kun ville kunne køre på Windows maskiner, modsat hjemmesiden som kan køre på alt. Hjemmesiden er udviklet ved brug af razor pages, da der her kan skrives i C#, som er et stort fokus punkt for semesteret, desuden gav dette også muligheden for at teste systemet, ved brug af NUnit.

5.3 Ekstern API

For at systemet kan leve op til kravspecifikationerne, skal der findes en database, som systemet kan tilgå, for at hente den nødvendige data. Der findes flere muligheder for at skaffe sådan en database, el. tilgå lign. data. F.eks. findes der sample-databaser som kan downloades fra nettet, gratis data samlet, og givet ud til netop test af prototyper og lign. Andre muligheder består af at denne type data kan hentes via REST API'er, som der er en del af på nettet. Nogen af disse API'er koster penge at tilgå, andre kan der laves non-profit aftaler med, og andre er helt gratis. Det de har til fælles er at de alle består af filmdata, og det som skildre dem er hvordan der kommunikeres med dem via en HTTP Client, og hvor detaljeret deres indhold er.

Følgende er en risiko analyse over vores muligheder, værende at enten læne op af en third-party API vs. at skabe/downloade dataene til en database.

Description	Prob. 1-5	Conseq. 1.5	Impact 1-5	Risk Mitigation Plan
TMDB stopper deres service	1	5	5	Download og gem data, i stedet for at trække på TMDB til alle datakald.
Databasen bliver for stor	2	1	2	Hold øje med størrelsen af DB, og sikre hvad det vil koste, og kræve af handlinger til at øge pladsen på serveren.
Fejl under download af TMDB data.	2	2	4	Hold øje med at dataene fra TMDB er korrekt, og test løbende.

Som det fremgår af risikoanalysen, er det en større risiko at bruge TMDB som ren API, og kun tilgå dataene efter behov, og et mere sikkert valg er at downloade den nødvendige data, og gemme det enten lokalt på serveren, el. på en ekstern MS SQL server. Den anden fordel ved at downloade dataene er at der kun skal downloades og gemmes den data, som Discover har behov for, for at leve op til kravspecifikationerne.

6 Projektbeskrivelse

I dette afsnit afdækkes den procestekniske del af projektforløbet, med særligt fokus på scrum-processen samt brugen af Redmines scrum-board funktionalitet.

6.1 Gruppeddannelsel

Gruppen er dannet på baggrund af et ønske om at holde ambitionerne inden for rimelige niveauer i forhold til de 5 ECTS-point der er tillagt semesterprojektet. Dette lykkedes fint og afgrænsningen af projektet blev overholdt og diskuteret ofte i plenum. Alle medlemmer i gruppen har tidligere arbejdet sammen med andre gruppemedlemmer, og der var derfor en generel forståelse af både fagligt niveau og ansvarsopfyldelse.

6.2 Process

Ved at der i dette projekt ikke er en klar fordeling mellem elementerne, som der i tidligere semesterprojekter har været med skilningen mellem hardware og software, var det vigtigt at have en struktureret måde at holde styr på processen og udviklingen af produktet. Ud fra de tidlige overvejelser om udviklingen af et Minimal Viable Product (se afsnit 3 - Kravspecifikation), blev det valgt at anvende en agil udviklingsmetode, scrum, der også blev brugt i semesterprojekt 3.

6.2.1 Scrum

Under udviklingsprocessen af softwaren var ønsket at der altid skulle være et funktionelt build af produktet. Dette stemmer overens med ønsket i under scrum at der efter hvert sprint er en iteration af produktet der kan præsenteres for en potentiel kunde. Scrum kræver under udviklingen af projektet mindre dokumentation hvilket er at foretrække når der anvendes teknologier der først bliver introduceret senere i semesterets undervisning, og den agile proces gør at disse teknologier kan implementeres når de bliver introduceret.

Derudover var det ønskværdigt at undgå merge conflicts, hvilket vil blive yderligere diskuteret under Git sektionen senere i rapporten, eller at der potentielt ville blive lavet dobbeltarbejde ved dårlig kommunikation omkring hvem der laver hvilke dele af det overordnede system. For at holde styr på de mange tasks der parallelt blev arbejdet på, blev det valgt at bruge et scrum-board som planlægningsværktøj. Da nogle gruppemedlemmer tidligere havde gode erfaringer, samt en afklaring om at alle ønskede funktioner var tilgængelige i værktøjet, blev Redmines Scrum-board valgt.

Projektets sprints blev planlagt tidligt i processen og en tidsplan blev udviklet. Tidsplanen, der kan ses på figur 2, var ment som en tidlig plan, hvor det vigtigste var at der blev afsat nok tid til både integration + accepttest samt rapportskrivning.

	Sprint	Uge
Opstart	1	7
Risikoanalyse		
Kravspeck		
Grov system arkitektur		
User stories		
Opsætning af tasks til sprintsne		
som afslutning		
Impl: Base solution	2	9
	3	11
	4	13
	5	15
PÅSKE		
	6	17
Implementering deadline	7	19
Integrationstests		
Rapport	8	21
Rapport	9	22

Figure 2: Tidsplan lavet i starten af processen

6.2.2 Redmine

Agile board			
Fjerde sprint (Apr 05 - Apr 25)		Charts	
<input type="checkbox"/> Filters		<input type="button" value="Add filter"/>	
<input checked="" type="checkbox"/> Apply <input type="button" value="Clear"/> <input type="button" value="Save"/> <input type="checkbox"/> Full screen			
New (0)		In Progress (0)	
Arkitektur i C4 modellen Oliver Spange		Pagination Andreas Steen Simmelkær Sørensen	
Frontend: Forsiden		production filter Rasmus Baunvig Aagaard	
jQuery Frontend: skift mellem foto-liste		Frontend: Enkeltfilm Anders Friis Søndberg	
Backend:Discover Algo Discover-Algo		Frontend: Discover resultater Anders Friis Søndberg	
Testing		Oprettelse af Jenkins Oliver Spange	
Forside/Søgning tests Patrick Elberg Larsen		Oprettelse af NUnit project Oliver Spange	
Dokumentation		year filter Rasmus Baunvig Aagaard	
Discover Refactor		CI Oliver Spange	
<input type="button" value="+ ADD NEW ISSUE"/>		Doc DB Andreas Steen Simmelkær Sørensen	
		Revenue & Budget filter Mads Kristian Key Høgh Christensen	
		Søge: ProdCompanies Patrick Elberg Larsen	
		Søge: Year Patrick Elberg Larsen	
		Log bog Anders Friis Søndberg	
		Doc Acquire Andreas Steen Simmelkær Sørensen	
		Søge: Persons Patrick Elberg Larsen	
		Frontend: Discover valg af film Oliver Spange	
<input type="button" value="+ ADD NEW ISSUE"/>		<input type="button" value="+ ADD NEW ISSUE"/>	
		<input type="button" value="+ ADD NEW ISSUE"/>	

Figure 3: Agile Board fra Redmine, her fra et af de senere sprints i processen

Scrumboardet, der kan ses på figur 3 har været et vigtigt værktøj til både at beskrive opgaver dybdegående og få en diskussion om hvad der reelt skal laves på projektet, samt en god måde at uddelegerere opgaver således alle ved hvad der laves og af hvem. Derudover giver det også god mulighed for at følge opgavers proces og sikre at der ikke glemmes opgaver. Redmines user experience har dog ikke været specielt god, og at oprette tasks + tilgå boardet har været en forhindring i forbruget af scrumboardet. Dette betyder også at under projektet blev aftaler også lavet udenom scrumboardet, da dette var langt mere tidseffektivt når det var en overskuelig mængde mindre komplicerede opgaver der skulle defineres og delegeres.

6.2.3 Git

I dette projekt har hele gruppen skulle arbejde på den samme applikation på samme tid. Dette kommer med en række udfordringer med hensyn til at kunne teste sin egen del løbende. Derudover skal der ved flere personers paralleludvikling sikres visse protokoller for at opfylde gruppens ønske om at bibeholde et fungerende build af produktet igennem hele processen. I undervisning er alle blevet fortrolige og komfortable med at anvende git og git-branches, hvilket blev vurderet til at være en oplagt mulighed for at undgå lokale ændringer af kode og undgå at 1 udviklers ændringer kolliderer med en anden udviklers ændringer.

Før udviklingen af en feature blev der oprettet en ny branch baseret på main-branchen, der altid skulle have et funktionelt build, således at nye features altid ville kunne blive testet på et system der fungerede. Disse branches blev udviklet på parallelt, og når en feature var færdigudviklet samt testet, blev branchen mergeret med main. Et eksempel på et merge med main kan ses på figur 4.



Figure 4: Screenshot fra github af merge af branch til main

En overvejelse omkring merge med main var at anvende pull requests for at få feedback på den nye feature, og for at sikre at der ikke lavet unødige ændringer til kodebasen der kunne skabe konflikter ved videreudvikling systemet. Dette blev fravalgt da det blev bedømt at implementeringen af unit

tests med continous integration via en Jenkins Server, især under udvikling af discover algoritmen, ville være tilstrækkeligt til at fange de fleste risici. Et udklip af Jenkins serverens brugergrænseflade kan ses på figur 5.

	Clean Workspace	Fetch from Git	Clean Build	Build	Test with coverage	Publish Test Results	Generate Coverage Report	Publish Coverage Results
Average stage times: (Average full run time: ~32s)								
#63 May 17 08:26	155ms	6s	2s	8s	16s	325ms	1s	399ms
#62 May 16 14:03	422ms	10s	4s	12s	11s	875ms	1s	468ms
#61 May 16 14:01	109ms	4s	1s	4s	9s	203ms	1s	328ms
#60 May 16 13:42	141ms	5s	2s	9s	26s failed	484ms		
#59 May 16 12:59	125ms	4s	1s	10s	26s failed	250ms		
#58 May 15 23:42	141ms	6s	4s	11s	11s	260ms	2s	375ms
#57 May 15 23:29	125ms	6s	1s	8s	14s	234ms	1s	469ms
#56 May 15 22:25	112ms	5s	1s	9s	30s failed	297ms		
	141ms	5s	1s	9s	14s	266ms	1s	375ms

Figure 5: Screenshot fra Jenkins, continous integration af projektet

Continous integration betyder at unit-tests for systemet bliver kørt ved hvert push af kode på main branch, og dermed undersøges der continuously at integrationen med den nye kode er kørt fejlfrit. CI kører udelukkende testsuiten under nye kodetilføjelse, og undersøger altså ikke om selve programmet reelt kan eksekvere. Derfor er det også nødvendigt, at der ved brug af CI bliver skrevet løbende unit tests til den udviklede kode, ellers bliver nye features ikke testet.

7 Arkitektur

7.1 Systemarkitektur

DiscoverMovies overordnede arkitektur kan ses på figur 6

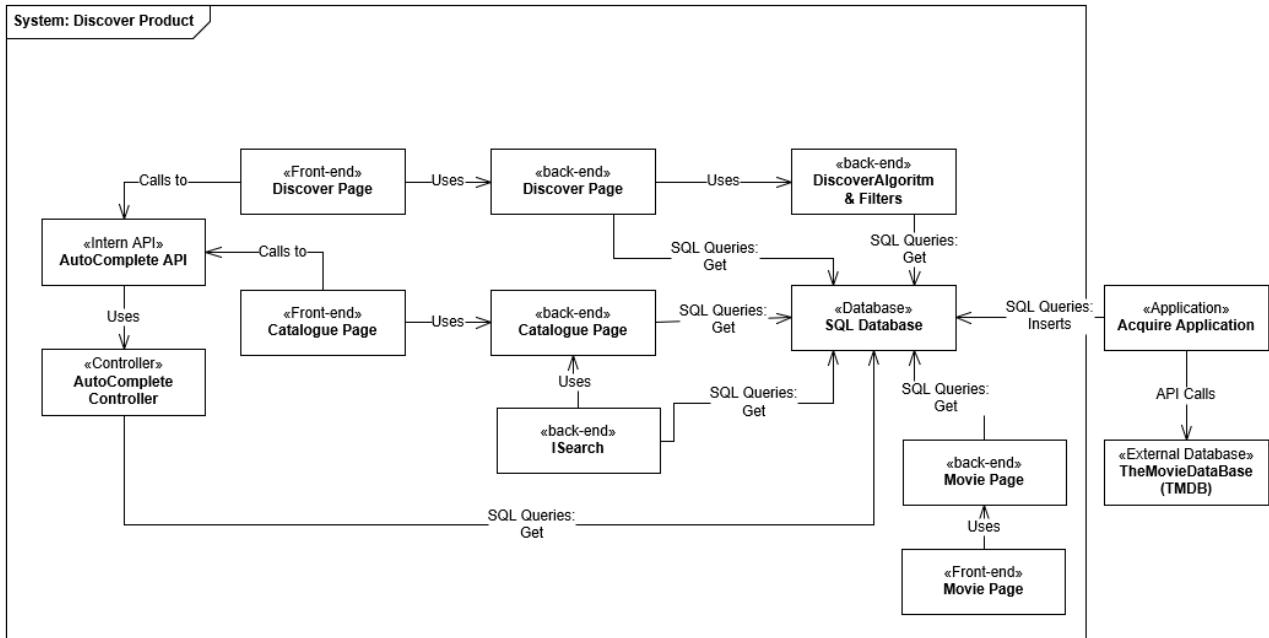


Figure 6: Den overordnede systemarkitektur. Læg især mærke til at Acquire ligger uden for systemets grænse.

DiscoverMovies indeholder en database (SQL Database på figur 6) der skal indeholde et stort udvalg af film, samt relateret information. Udvælget af film skal udvindes eksternt for at undgå manuel data entry, og derfor trækkes der på en ekstern database der ikke indgår i systemet.

En applikation (Acquire Application på figur 6) der foretager database queries for at hente data fra den eksterne database (TheMovieDataBase på figur 6) og indsætter denne data i den interne database, SQL Database.

DiscoverMovies består overordnet af 3 sider, samt den baglæggende backend med selve funktionligheten af disse sider.

Movie Page indlæser data fra databasen og præsenterer denne data på en enkeltside.

Catalogue Page er applikationens front page / indexpage og indlæser også data direkte fra SQL Database og fremviser udvalget. Søgefunktionaliteten af Catalogue Page anvender interfacet ISearch, som der er 3 implementeringer af. ResolveSearch implementerer ISearch og kalder så fra databasen og opdaterer derefter catalogue page.

Discover Page, der er den primære og vigtigste feature i produktet, har størstedelen af dens funktionlighed samlet i en samlet algoritme (DiscoverAlgorithm & Filters på figur 6), der anvendes af Discover Pages backend.

Både Discover Page og Catalogue Page anvender i deres søgefelt autocomplete for at øge brugervenligheden. Autocomplete er udviklet som en lokal API, der via et js-script i begge siders front-end, kalder AutoComplete Controller der trækker på databasen for at få forslag til færdiggørelse af søgninger.

7.2 Softwarearkitektur

For at formidle software arkitekturen, bruges C4 modellen. Denne model formidler arkitekturen af produktet på 4 niveauer (Context, Containers, Components, Code). Ved at benytte denne metode, bliver produktet først formidlet i en abstrakt og overordnet form, og til sidst i en mere detaljeret og bestemt form. Diagrammerne bliver mere detaljeret, da de sidste niveauer af C4 blot analyserer en node fra det tidligere abstraktionsniveau.

7.2.1 System Context niveau

På figur 7 vises Context niveauet af systemet, hvilket er det højeste abstraktionsniveau. Her er det tydeligt, hvilke udenfra aktører der behandler eller bliver behandlet af vores system. En bruger benytter DiscoverMovies systemet, ved at tilgå hjemmesiden og dens funktionalitet. DiscoverMovies bruger TMBD, da systemets interne data er defineret fra den.

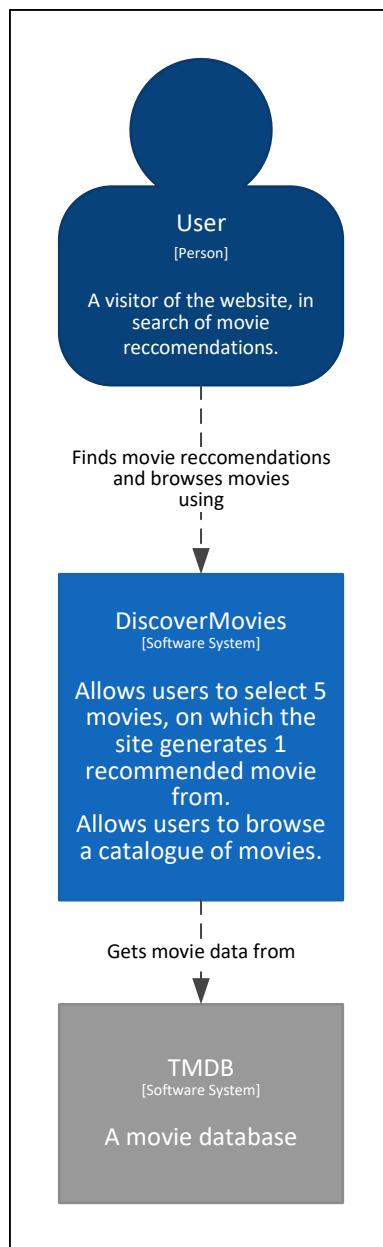


Figure 7: C4-modellering af system - system context niveau

7.2.2 Container niveau

På container niveauet af C4 (på figur 8) vises nu de 4 containers som DiscoverMovies systemet består af. Brugeren vil interagere med en Web Application lavet i Razor Pages. Denne razor applikation vil operere server-side, og servicere brugeren med HTML sider, afhængigt af hvilken URL brugeren forespørger om. Disse HTML sider vil have noget funktionalitet, som laver REST-API kald på serverens API-Controller - hvori kaldende håndteres og data tilbagesendes til brugeren. Denne API-Controller container vil derfor være knyttet til systemets database, hvori filmdata er lagret.

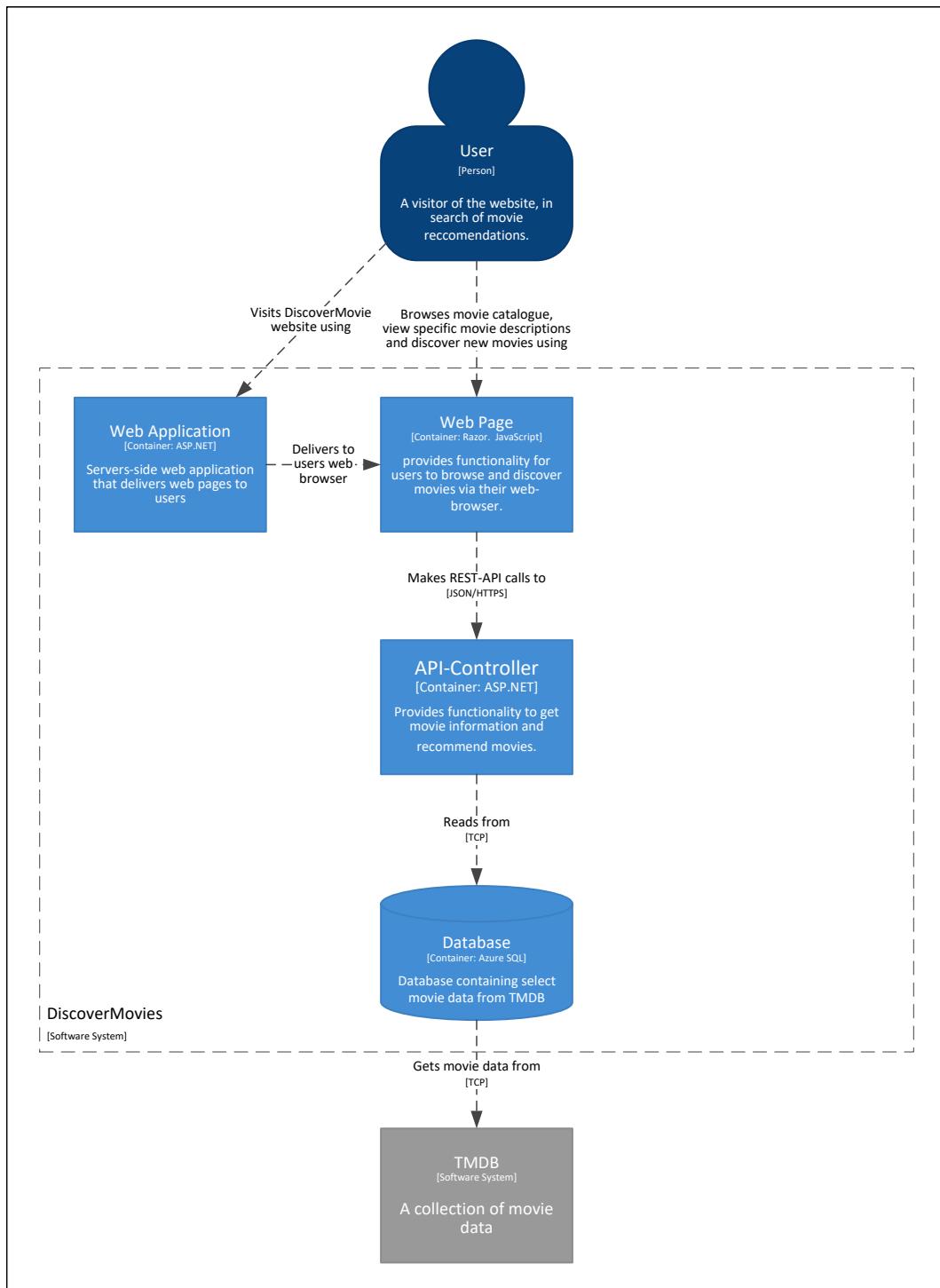


Figure 8: C4-modellering af system - container niveau

7.2.3 Component niveau

Ved component niveauet, detaljegøres containers, som blev vist på figur 8. Der vil herunder kun blive vist 2 containers på component niveau (Web Page og API-Controller), da kun disse containers har indhold, som er værd at detaljegøre til komponenter.

På figur 9, vises komponenterne, som Web Page består af. Heri er de 3 sider (Discover Page, Catalogue Page og Movie Page), som indeholder cshtml, hvilket bliver renderet som html, når det renderes ved brugeren. Der er også en javascript komponent, som bruges til at lave autocomplete på søgninger.

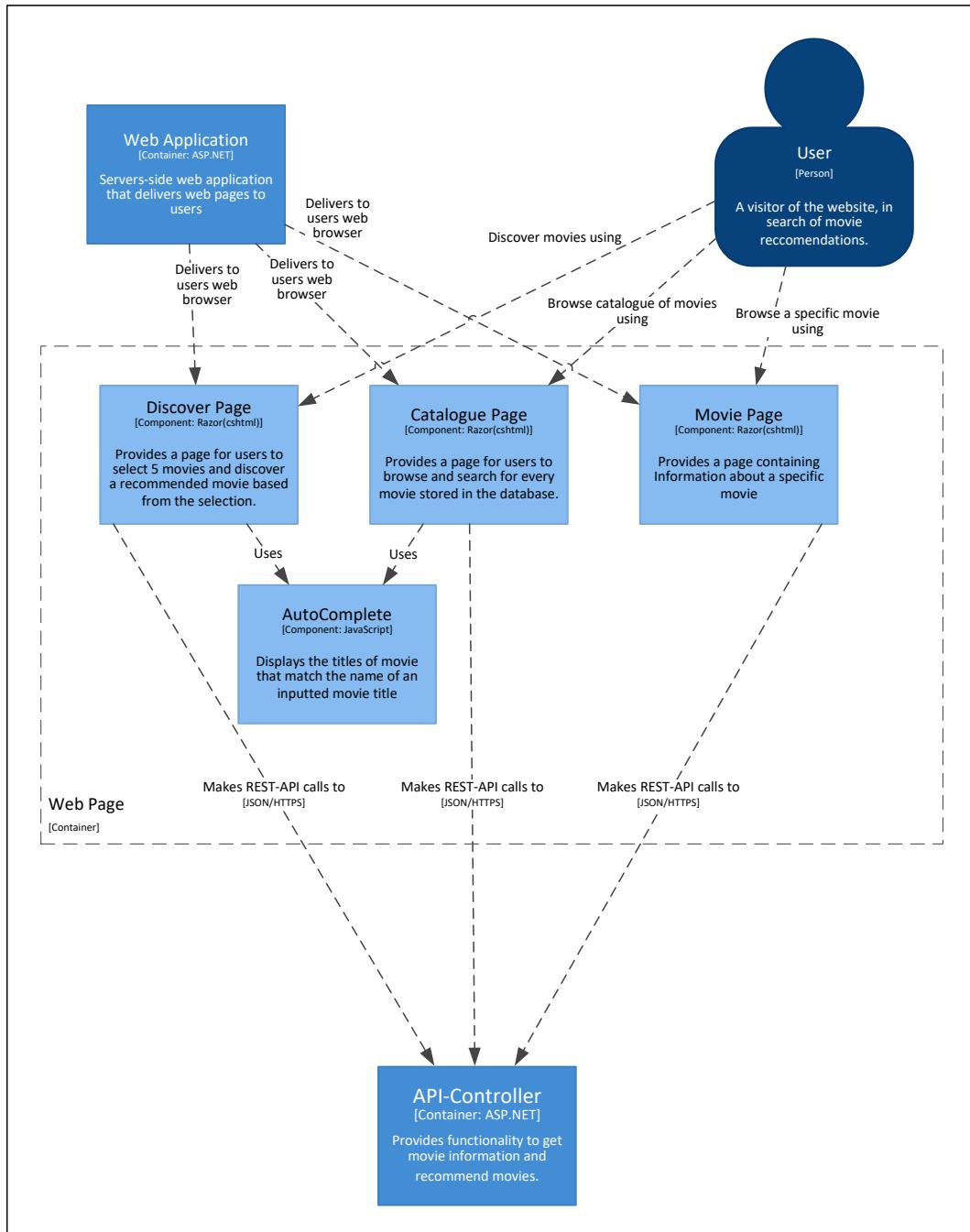


Figure 9: C4-modellering af system - Web Page på component niveau

På figur 10, vises komponenterne, som API-controlleren består af. Heri er der controllers til hver Razor Page side (Discover Page, Catalogue Page og Movie Page), som håndterer de REST-API kald der er knyttet til den pågældende side. Discover Page siden bruger også en Discover controller, som indeholder algoritmen til at anbefale en film til brugeren. AutoComplete controller er knyttet til AutoComplete komponentet i figur 9, denne controller håndterer autocomplete søgningerne og returnerer foreslagene. Search Controller bruges af Catalogue Page til at håndtere søgninger givet et vilkårligt filter.

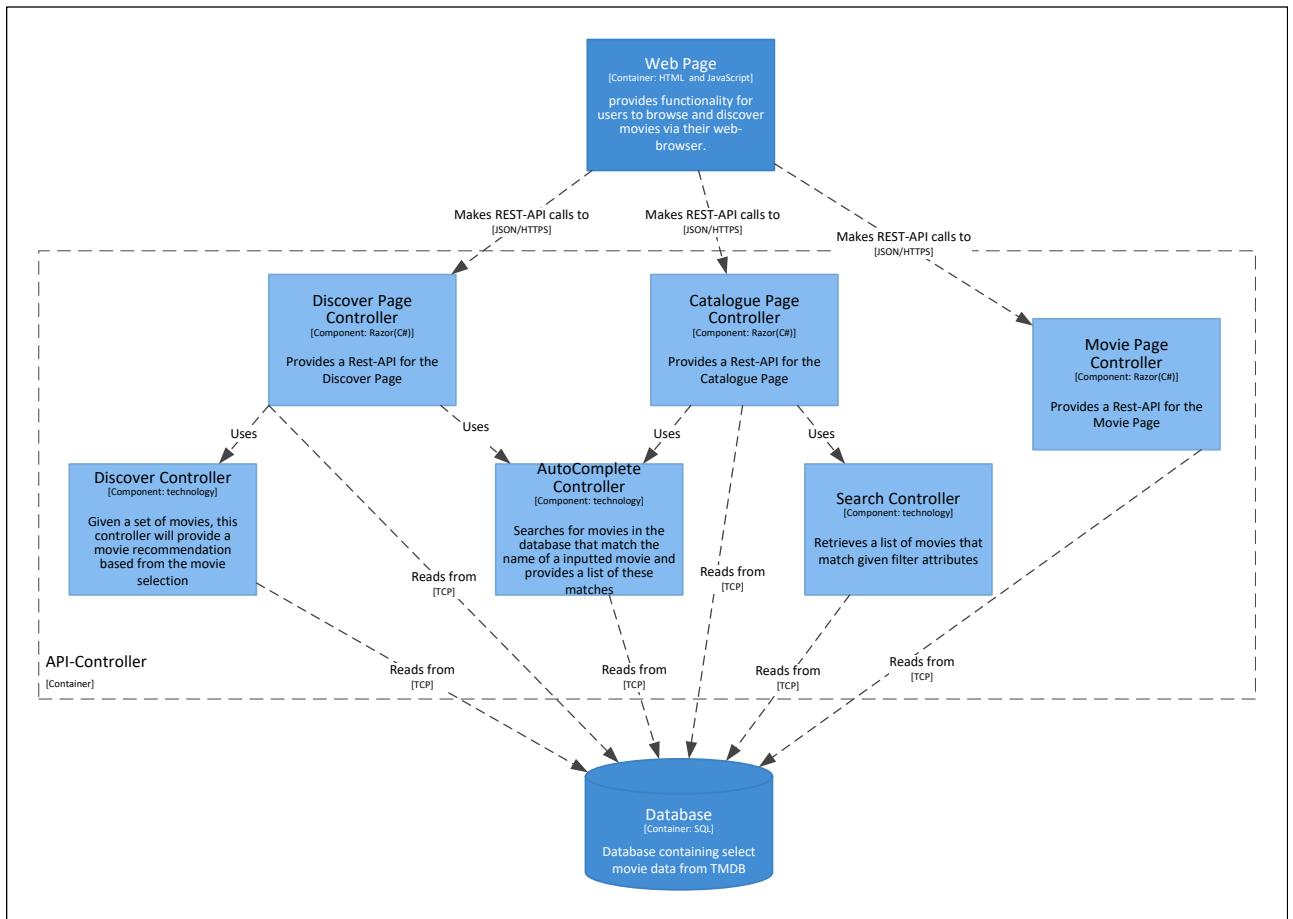


Figure 10: C4-modellering af system - Api-Controller på component niveau

7.2.4 Code niveau

På dette niveau udpenses de klasser og tildels designet af de komponenter fra det forrige niveau. De nedenstående diagrammer fokuserer på 2 af komponenterne fundet i diagrammet på figur 10, hvilket er Discover Controller og Search Controller. Dette skyldes at disse controller komponenter er komplekse nok, til at diagrammerne her kan give værdi til udviklingen og formidlingen af dem.

På figur 11 vises de klasser, som Discover Controlleren vil bestå af. Discover komponentet vil modtage 5 film, hvor at hver films attributter vil gennemgå et filter, som prioriterer nye film baseret på lighederne med de originale film. Der er i systemet 6 filtre, som i algoritmen opdaterer prioriteringen på de film som kan anbefales til brugeren. Til sidst vil den højst prioriteret film blive returneret til brugeren. Til denne løsning bruges Visitor design patternet - der kan læses mere om på afsnit 9.3.2.

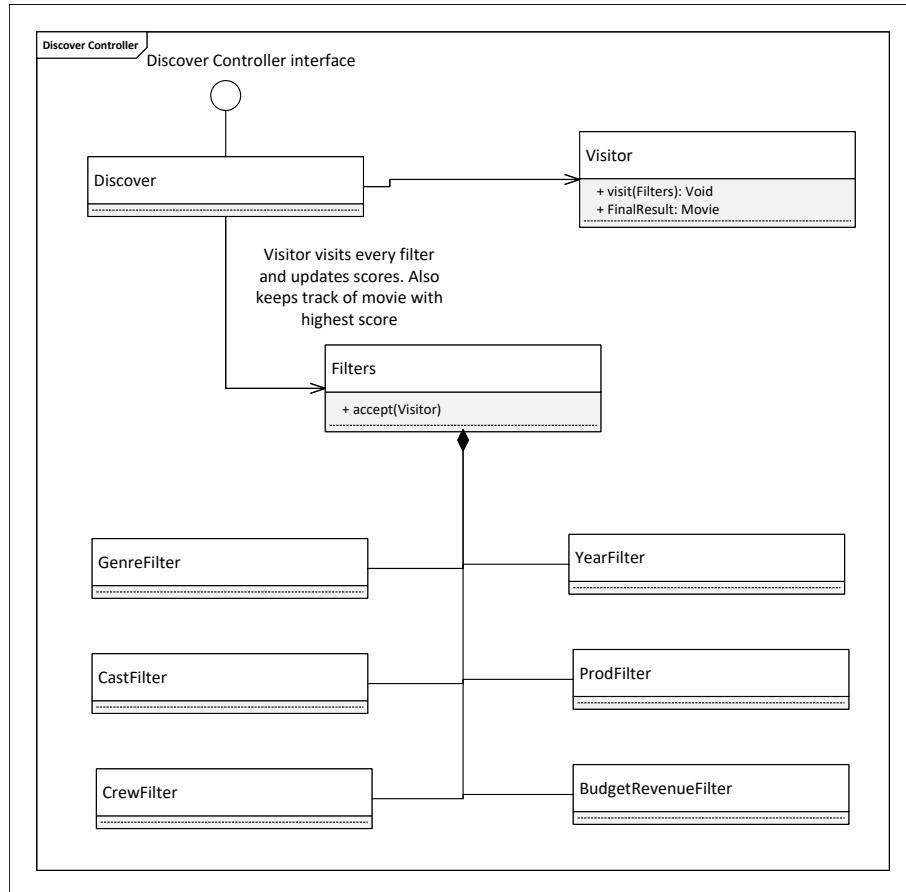


Figure 11: C4-modellering af system - Discover på code niveau

På figur 12 vises et klasse diagram af search komponentet. Dette komponent bliver beskrevet i større detaljegrad i afsnittet 9.2.2. Dette komponent har til funktion, at returnere en liste af film, der afhænger af de parametre som er sat for søgningen. Search komponentet vil benytte strategy pattern, til at skifte strategi afhængigt af hvilken søgning der laves på filmene. Det betyder at der skal implementeres en klasse for hver strategi - alle strategi klasserne implementerer det samme interface, for at gøre dem udskiftelige.

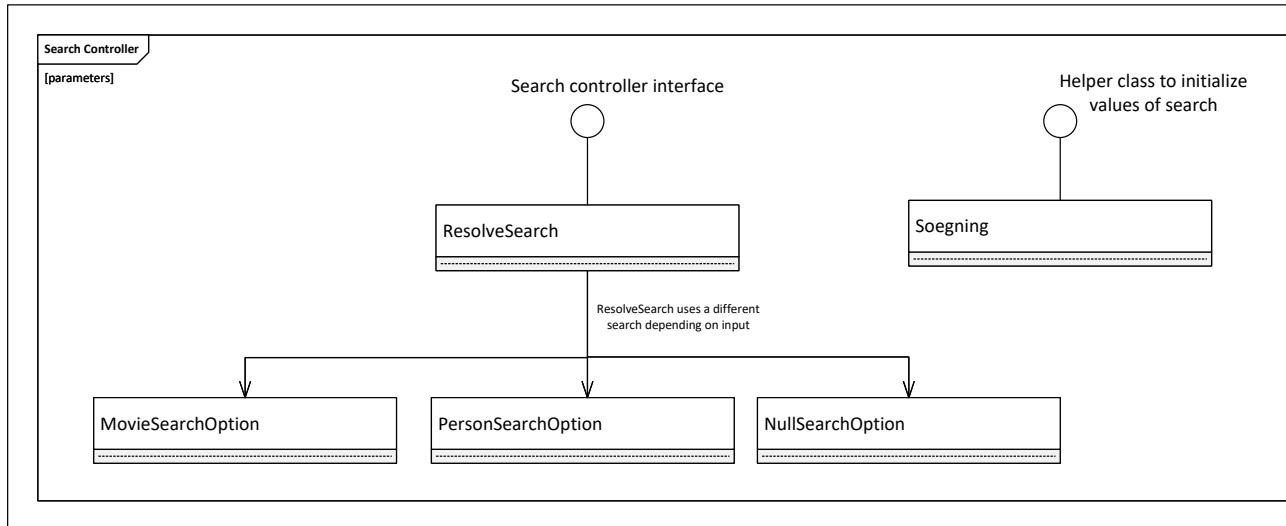


Figure 12: C4-modellering af system - Search på code niveau

8 Analyse af Software

8.1 Discover Algoritme

Discover algoritmen er blevet udviklet gennem flere iterationer, og vi har gjort os en masse tanker om hvordan den skulle struktureres og virke. For strukturen startede vi med en ide om at hvert filter skulle være deres egne dele der så kunne samles. Dette var et valg vi tog for nemt at kunne tilføje nye filtre uden at skulle ændre i en stor proces, og samtidig for at kunne udvælge hvilke filtre der skulle bruges i fremtiden, da vi havde en videreudvikling planlagt hvor man ville kunne låse filtre. Det var et godt valg da vi endte med at tilføje et nyt filter, og det havde været betydeligt mere besværligt uden. For at opnå dette skulle vi definere et input og et output format som alle filtrene skal følge. Dette blev gjort ved at lave en shortlist af filmobjekter til input, og en liste af samlede filmobjekter og en score, som så kan behandles for at give et output baseret på alle de ønskede filtre. I anden iteration har vi valgt at lægge endnu mere vægt på strukturen, og har her brugt visitor design pattern til at samle filtrene under en endnu mere organiseret struktur.

Vi har taget nogle valg på hvad vi filtrerer på og hvordan vi udvælger de parametre hvert filter skal bruge. Vi har så vidt muligt forsøgt ikke at lege ”smagsdommere” på hvad der er en god film. Der har dog været valg vi har skulle tage for at optimere koden, men også for at undgå usammenhængende resultater. Dette kommer til syne ved crew filtret. Der er en lang liste af crew til de fleste film, men det er ikke alle der har haft et input på filmen, det er derfor valgt kun at tage crew med kreative roller. Valget af filtre er baseret på tilgængeligheden af data, og relevansen af parametre. Vi har valgt at fokusere på de dele vi mener er mest relevante. Dette er igen et sted hvor vi har skulle tage valg om hvad der er vigtigt for en film af praktiske årsager. Vi har valgt dem ud fra at det er de elementer der vil have en tydeligere effekt på filmen, og derfor vil være et forslag en bruger vil kunne se ligner brugerens 5 valgte film.

8.1.1 Filtre

Genrefilter

Genrefiltret tildeler point til de film der har genrer der er delt med de fem udvalgte film.

Crewfilter

Crewfilteret tildeler point baseret på hvor mange i crewet der er gentaget. Der sorteres så det kun er relevant crew der bliver medtaget.

Castfilter

Castfiltret tildeler point baseret på hvor mange skuespillere der går igen imellem filmene. Der sorteres også her efter relevans så kun de mest populære skuespillere medtages.

Yearfilter

Yearfiltret tildeler point til de film der er udgivet i et årstal der er tæt på de givne film.

Productionfilter

Produktionsfiltret tildeler point ud fra hvor mange produktionsfirmaer der går igen mellem filmene.

Budgetandrevuefilter

Budget og indtjeningsfiltret tildeler point ud fra om de er profitable, om deres budget falder tæt på gennemsnittet for de givne film, og ud fra en inddeling for deres budget i form af et kvartilsæt.

8.2 Search Bar

Igennem udviklingen af DiscoverMovies, er der blevet drøftet idéer til hvordan søgefunktionen skulle virke. Igennem de tre iterationer af searchbar har koncepterne af søgerfeltet og søgerbaren ikke ændret

så meget udseende, men den underliggende funktionalitet har ændret sig en del. Ideelt set skulle søgerfeltet selv kunne detektere, hvad at der blev søgt på, sådan at brugeren ikke skulle tage stilling til for mange parametre. F.eks. på imdb.com hvor søgerfeltet selv finder ud af, hvad brugeren søger efter. Dette virke intuitivt. Dog vil det give en dårligere brugeroplevelse, hvis at brugeren ønsker at søge specifikt. En dynamisk søgerbar ville samtidig blive en stor mundfuld inden for rammerne af projektet.

I første iteration af Searchbar, blev den grundlæggende struktur lavet, som dannede grundlag for Searchbar. Det at søgerfunktionen håndterer flere parametre, som både kan være sat, og andre som ikke er sat, var en udfordring. Fordi søgerfunktionen skulle matche variabler valgt af brugeren op imod databasen, samtidig med ikke alle søgerparametre ville behøves at være sat. Brugeren skulle som udgangspunkt have frie valgmuligheder for søgning af film, hertil muligheder for Searchinput (Input fra søgerfelt), Genre, Searchtype (Searchinputs type) og Year. I den første iteration blev der fokuseret på at få Genre, og Søgerfeltets funktionalitet til at virke, med opslag op imod databasen, hvilket lykkedes. På dette stadie var der ret meget kode i Code-Behind filen af catalogue page med OnGet() og OnPost() metoderne. Der var samtidig lavet checkboxe, som brugeren skulle tjekke af om for at søge på en genre(valgt i en dropdownmenu) eller en titel af film i søgerfeltet. Denne logik var på dette tidspunkt fint, men skabte problemet, at vi ikke kunne krydssøges for filmtitel, og en bestemt genre.

Der var på dette tidspunkt i frontend delen lavet et fornuftigt søgerfelt samtidig med visning af et katalog af film. Der er en submit-knap til brugeren, som sørgede for at sætte gang i en søgning.

Figure 13: Screenshot af Searchbar

I anden iteration af searchbar, blev der fokuseret på at lave code-behind filen, sådan at den blev klar til flere søgerparametre som year, eller produktionsselskaber, og den eksisterende struktur af if-statements, blev der bygget videre på. Der blev brugt en del tid på at få logikken til at virke, som resulterede i at, koden måtte refaktoreres, for både at gøre den mere testbar, og overholde SOLID principper. Foruden at logikken var blevet uoverskuelig, så var flere af databasekaldene lavet i foreach loops, som resulterede i rigtig lange søgertider. I den første iteration af searchbar, var der lavet checkboxe til en bruger i stedet for dropdown menuer. I enighed med gruppen, så blev der lavet dropdown menuer til genre og year i stedet for. Anden iteration blev herefter afbrudt, da der var ikke var grund til at gå videre med den daværende struktur, som vil kræve en større refaktorering.

Den tredje iteration startede ved at refaktorere koden, og implementere flere dele med strategy design pattern. Dette mønster passede godt ind, da man indkapslede klasserne, samtidig med at man gjorde dem udskiftelige. Der lavet to interfaces ISearch og Iinitializer.

Initializer var tiltænkt som interfacet for initialisering af dropdown menuerne for hhv. genre, year, og searchtype. Her ville klassen "søgning" arve interfacet og initialisere menuerne i hhv. OnGet() og OnPost(). Interfacet kunne være blevet brugt i tre underliggende klasser for hver deres initialisering, men ved det var samme grundlæggende funktion, så blev de slået sammen, som egentlig forbrød sig imod strategy design pattern.

ISearch interfacet laver to metoder SearchInput() og Setattributes() med søgertypen, input fra søgerfelt, year, og genren som input parametre. Setattributes() blev lavet for at kunne teste parametrenes tilstand om de var sat korrekt, dette blev suppleret af public getter til hver attribute, for at aflæse de faktiske værdier sat fra brugeren.

SearchInput() Metoden filtrerer og efterspørger data via et kald til databasen ud fra værdierne sat i parametrene af brugeren. Dette sørger for minimal forespørgsel hos databasen, men desværre blev

metoden svær at teste. Dette er fordi, at der sker to ting samtidig, og ved man bruger MyDbContext, som arver fra interfacet DbContext. Dette kunne være opdelt i to klasser, så en klasse hentede alt data, og anden klasse filtrede dataen. Dette virkede som i overkanten for at teste en klasses metode. Man ville måske nemmere kunne teste klassen med frameworket Xunit, men da viden og undervisning er minimal indenfor dette framework, synes vi det var et stort skridt at tage. Derfor endte vi med at lade klassens metoder blive.

En udfordring var at detektere hvad teksten af søgerfeltet i searchbar skulle søge efter. Ved at man kan søge efter film med en bestemt person, som er i en anden tabel end titler på film. Der blev lavet en Searchtype option dropdown menu, som giver brugeren valget om det er, filmsøgning (klassen MovieSearchoption), en personsøgning (klassen PersonSearchoption), eller om brugeren søger efter film baseret på genre og-eller år (klassen NullSearchoption). Klasserne blev lavet under interfacet af ISearch. Searchoption aktivere ”søgefeltets tekst”. Ud fra en brugers perspektiv, kan det virke irriterende, at man skal vælge en searchoption før man kan bruge tekstfeltet. På den anden side kan man sige, at det samtidig gør brugeren bevist, om hvad de søger efter, og sørger for søgningen er specifik efter brugers søgning.

Generelt er searchbaren blevet implementeret som vi ønskede, man kan søge på kryds af de forskellige parametre. Dette var det største krav for searchbaren som er opfyldt. Vi kunne godt have tænkt os, at have haft mere tid til at undersøge muligheden for at søge lidt mere dynamisk i søgningen, sådan at søgerfeltet, selv kunne detektere hvad man søger efter. Derudover kom vi ikke i mål med pagination, som er opdeling af søgesiden på flere pages.

8.3 Moviepage

Den specifikke filmside, er en razor page, som anvender databasen, til at vise alle forudbestemte informationer, der er blevet valgt er relevante, for de enkelte film. I sidens code behind hentes informationerne fra den valgte film ned i en liste, gennem databasen, baseret på et id der sendes med igennem URL’et. Informationerne fra denne liste bliver vist på siden, de relevante steder, sammen med et coverbillede af filmen, der hentes fra TMDB’s API.

Informationen bliver hentet ind på siden, ved brug af razor pages funktion til at skrive C# kode i cshtml filen. Informationen fra code behind filen bliver udskrevet de rigtige steder i cshtml filen ved brug af @ operatoren. Ved brug af razor pages @page direktiv i cshtml filen, bliver filen til en MVC (Model-View-Controller) action, hvilket betyder at den håndterer requests direkte uden at gå igennem en controller, hvilket også betyder at der kun er en model og et view fra MVC patternet, som gør projektet mere simplificeret.

8.4 Database Struktur

For at kunne vælge strukturen for databasen, har de primære overvejeler været omkring hvilken type data der skulle bruges til enten at kunne søge på, eller vise på enten Catalogue, Movie eller Discover Page. F.eks. til Catalogue bruges kun titel, og poster URL, til visningen, men til søgningen skal der bruges titel, udgivelsesår, personer medvirkende (foran og bag-kameraet), samt genres. Det vil sige at de forskellige features og sider har forskellige behov. Den anden overvejelse har været hvordan dataene skal være associeret med hinanden, f.eks. hvordan tilgås nemmest og mest effektivt den data som der er behov for. Har man et film ID, hvordan findes så alle personer medvirkende i den givende film. Disse problemer er blevet overvejet, hvor resultatet, gennem flere iterationer, har resulteret i designet, der kan ses på figur 14.

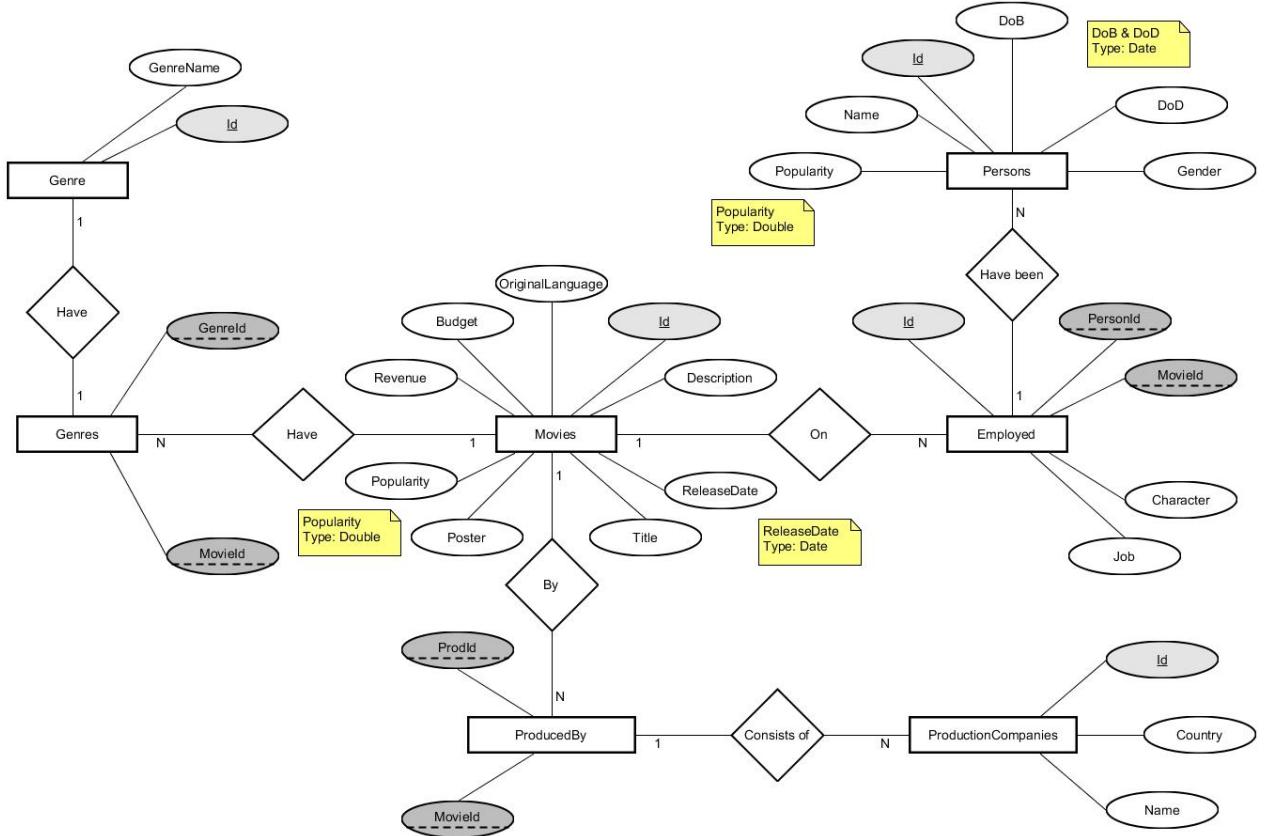


Figure 14: Endelige database schema

8.5 Acquire Application

For at kunne have et fungerende system, hvor der både kan søges i film, men også gives film forslag, så skal der forfindes en database. Det vil sige at der skal enten findes en database eller skabes en fra bunden. Givet at en komplet database skal findes, skal den være indholdsrig nok til at tillade funktionerne som at udføre søgninger, og give forslag fungere, og leve op til krav for systemet. Dette kan være svært, og søgninger på nettet, trods resultater, har kun leveret små sample-size databaser, som gruppen vurderede til ikke at være tilstrækkelig.

Den anden mulighed er at trække direkte på en API-service som allerede eksisterer, og som er vedligeholdt, for ikke at tale om at indeholde et omfangsrigt katalog. En undersøgelse af muligheder har vist at der findes op til flere af sådan slags løsninger. Nogle koster penge, enten som abonnement, el per request, ingen af delene noget som gruppen har råd til, som med det samme kasserer de løsninger. Der er dog også nogle filmdatabase API'er som er gratis, enten siden deres indhold er crowd-sourced, el. fordi der ingen garantier er for deres oppetid mm. Vigtigt for samtlige API'er undersøgt, er at de skal leve op til systemets krav for typer af indhold, f.eks. skal der forfindes information om hvem har medvirker, arbejdet bag kameraet, udgivelses dato mm. for at dataene er brugbar.

Blandt dem som er blevet undersøgt er:

- Internet Movie Database
- The Movie Database
- unofficial Netflix online Global Search
- The Open Movie Database API

Trods de alle er gratis, så er tilgangen til dem meget forskellig. IMDB har en længere godkendesproces, og uNoGS har utroligt dårlig service ifht. oppe-tid mm. og så er deres data begrænset til kun film fra Netflix's service. The Open Movie Database bliver også kasseret pga. begrænset dataindhold. Så vinderen bliver TMDB, siden den er gratis, har en kort godkendelsesproces, og den har et datarigt, og omfattende katalog.

Som det også blev vurderet i risikoanalysen, så er det mere hensigtsmæssigt at downloade indholdet fra en API service, i stedet for at trække på den "live" version. Der skal derfor produceret en applikation, separat fra selve Discoversystemet, som kan bruges til at hente alle nødvendige data, samt indsætte det i produktionsdatabasen.

8.6 NUnit

For både at leve op til krav for semesterprojektet, men også sikkerhed for integriteten af kodebasen under udvikling, tages CI (Countinuous Integration) i brug. Til det formål skal der vælges et test-framework, og siden systemet primært er udviklet i C#, er det oplagt at tage enten NUnit eller XUnit i brug. Da det kan være en omfattende opgave at have grundig tests for hele kodebasen, og da systemet udvikles i både alm. C# og Razorpages, ville begge frameworks være aktuelle. For at holde arbejdsindsatsen for tests nede, men stadigvæk sikre at grundfunktionaliteten er grundigt testet, er det blevet besluttet kun at vedligeholde CI for selve Discover forslagsfunktionen. Dette betyder at det kun er nødvendigt at tage NUnit frameworket i brug.

Trods NUnit er valgt, er der sidevalg som også skal besluttes, som hvilket Mock framework der skal bruges foruden NUnit. Da de fleste i gruppen har bedst erfaringer med Moq, bliver det valgt frem for NSubstitute.

9 Design og implementering

9.1 Indledning

Dette kapitel vil gennemgå de væsentlige dele af designet og implementationen af systemet. Det meste af udviklingen har været delt op enten mellem personer i gruppen, el. mindre teams i gruppen, disse forskellige features er derfor opdelt lign. her.

9.2 Catalogue

Catalogue page er lavet som en razorpage. Derfor er begreberne backend og frontend lidt flydende. Backend delen består af en code-behind fil, som tilgår nødvendige klasser og interfaces. Frontend delen er primært skrevet ved at bruge HTML, CSS, og lidt javascript. Yderligere dokumentation, kan ses i bilag afsnittene searchbar på bilagsnummer A.4.

9.2.1 Implementering Frontend

Frontend af Searchbar består primært af en HTML Form, en række dropdown menuer, en submit knap, og en reset knap som danner grundlag for selve søgerbaren. Her er søgerfeltet et input felt, som binder sig til code-behind klassen InputMovie. Selve dropdown menuer binder sig code-behind properties, som bliver initialiseret i Iinitializer. Søgebaren kan ses på figur 13. Selve det som danner det visuelle katalog, er et forloop hvor der laves en ”firkant”, hvor hvert filmbillede og filmtitel bliver indsat. Dette er på baggrund af logikken i en property fra code-behind filen Movielist. På baggrund af brugerens søgerparametre, kommer der film i denne liste. Hvis der ikke er noget billede på filmen, indsætter den vores eget ”Missing Poster” billede, og hvis ingen film matcher søgning, så kommer der også en fejlbesked frem til brugeren om dette. Se figur 15

```
<div class="row">
    @foreach(Movie movie in Model.MovieList)
    {
        <a href="/movie/@movie.movieId" class="col-xl-2 col-sm-6 col-md-4">
            @{
                if(@movie._posterUrl == null)
                {
                    @movie._title <br/>
                    <img src=/Images/MissingPoster.png height="237" width="165">
                }
                else
                {
                    @movie._title <br/>
                    <img src=@("https://www.themoviedb.org/t/p/w220_and_h330_face" +
                    @movie._posterUrl) height="237" width="165">
                }
            }
        </a>
    }
    @if (Model.MovieList.Count == 0)
    {
        <p> Sorry, no movies is matching your search options...
        <br/>
        Please try changing the search options, and check for misspelled words.
    }
</div>
```

Figure 15: Frontend Implementation Searchbar

9.2.2 Implementering Backend

Backend i searchbar er code-behind filen med tilhørende interfaces og klasser. I code-behind filen, er der lavet to metoder som er hhv. OnGet(), som indlæses når en bruger loader DiscoverMovies. Der er også lavet en OnPost() metode, som bliver aktiveret når en bruger i frontend'en trykker på submit knappen. Her vil den returnere Movielisten med film filtreret med parametrene fra brugerens

søgning ved at bruge databasen. Det meste af backend for Search blev lavet i 3. iteration. Strukturen var lavet i de forrige iteration, men ikke tænkt igennem i forhold til strategy pattern. Strukturen af klasser, og interfaces kan ses på figur 16

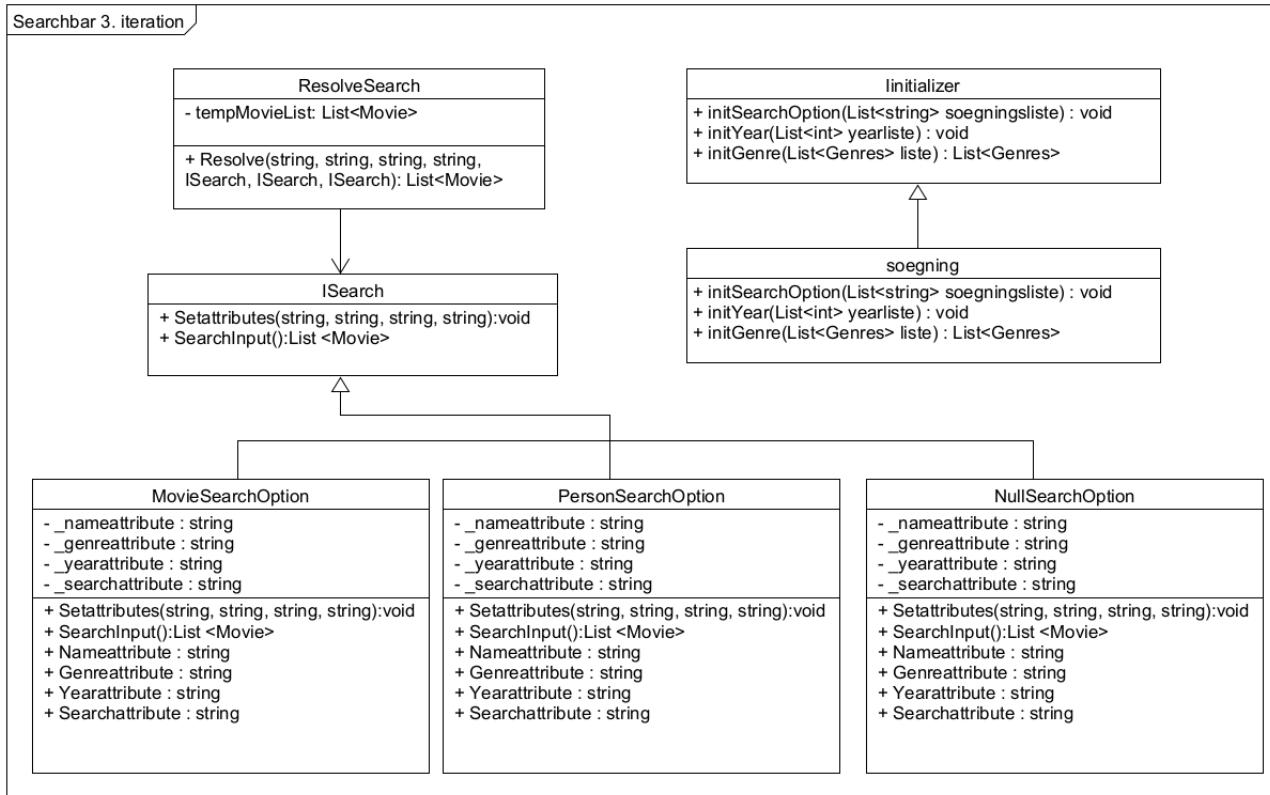


Figure 16: Searchbar klassediagram

ResolveSearch består af 3 if statements. Den sidste string i resolve metoden, er Searchtype, som egentlig detekter hvilken SearchOption, som bliver brugt. Dette er noget brugeren vælger i frontend delen. Initializer og soegning klassen er del af discovermovies, men bliver primært brugt i hhv. OnGet() og OnPost() metoderne, og bliver initialiseret hvergang at metoderne bliver kaldt.

9.3 Discover

Discover-Forslags feature er delt op i to dele, Frontend, og Backend. Frontend dækker selve Razor siden som en bruger vil kunne besøge på siden, samt dens code-behind fil, hvor backend delen er en serie af klasser som arbejder tæt sammen. Frontend og Backend arbejder altså meget tæt sammen for at leverer Forslags-feature.

9.3.1 Implementering Frontend

Razor siden for Discover Page, indeholder en form, hvori man kan lave en søgning på en film og sekventielt tilføje film til listen af parametre for discover algoritmen. Hver gang en film tilføjes, vil siden genindlæses og vise de tilføjede film. Logikken for denne operation vises som et sekvensdiagram på figur 19. Her vises der hvordan AutoComplete komponenten bruges til at hjælpe brugeren med foreslag til søgningen. Hvis der er 4 film i parameterlisten, og der tilføjes endnu en film, så vil discover algoritmen blive eksekveret. Resultatet af discover algoritmen vil blive tilføjet til brugerens klient, efter klientens siden genindlæses.

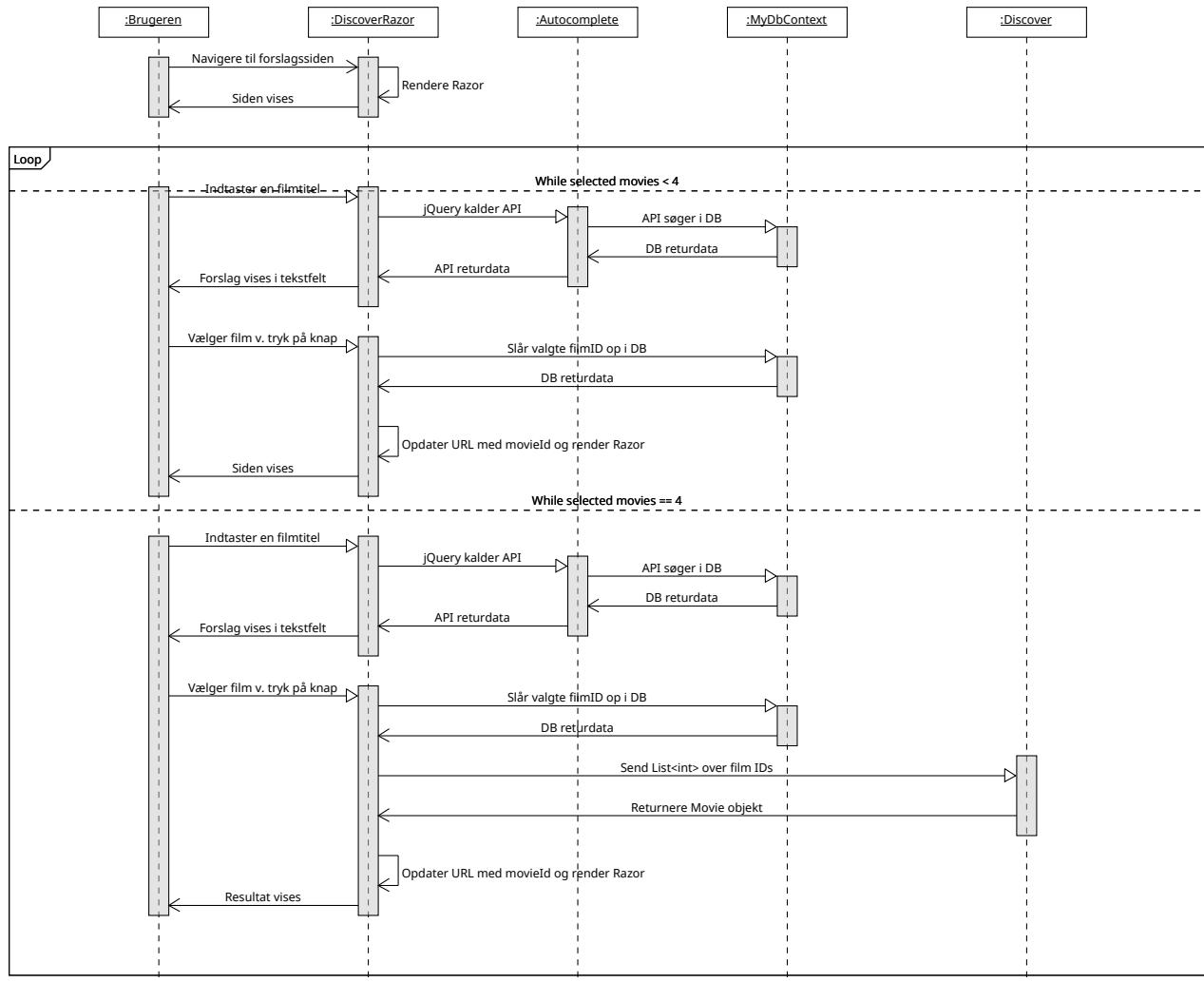


Figure 17: Sekvensdiagram over Discover-frontend

For at lagre filmene efter hver genindlæsning, gemmes movieID'er i klientens URL, og bliver indlæst derfra efter hver genindlæsning. Denne løsning, til at gemme data, vælges grundet simpliciteten af implementeringen, andre mere komplekse løsninger ville være gemme dataen i Cookies eller Session State.

9.3.2 Implementering Backend

Backend delen af Forslags-featuren består af en stribe klasser som opgør denne feature. Det var ikke åbenlyst for Discover-teamet i starten, hvilken struktur eller Design Pattern at der kunne anvendes, men efter at have reflekteret over en velfungerende prototype, i slutningen af første iteration, blev det langt nemmere at se at et Visitor Design pattern kunne tages i brug. For at danne et overblik over alle klasserne, kan man beskue dets klasse diagram, se figur 18.

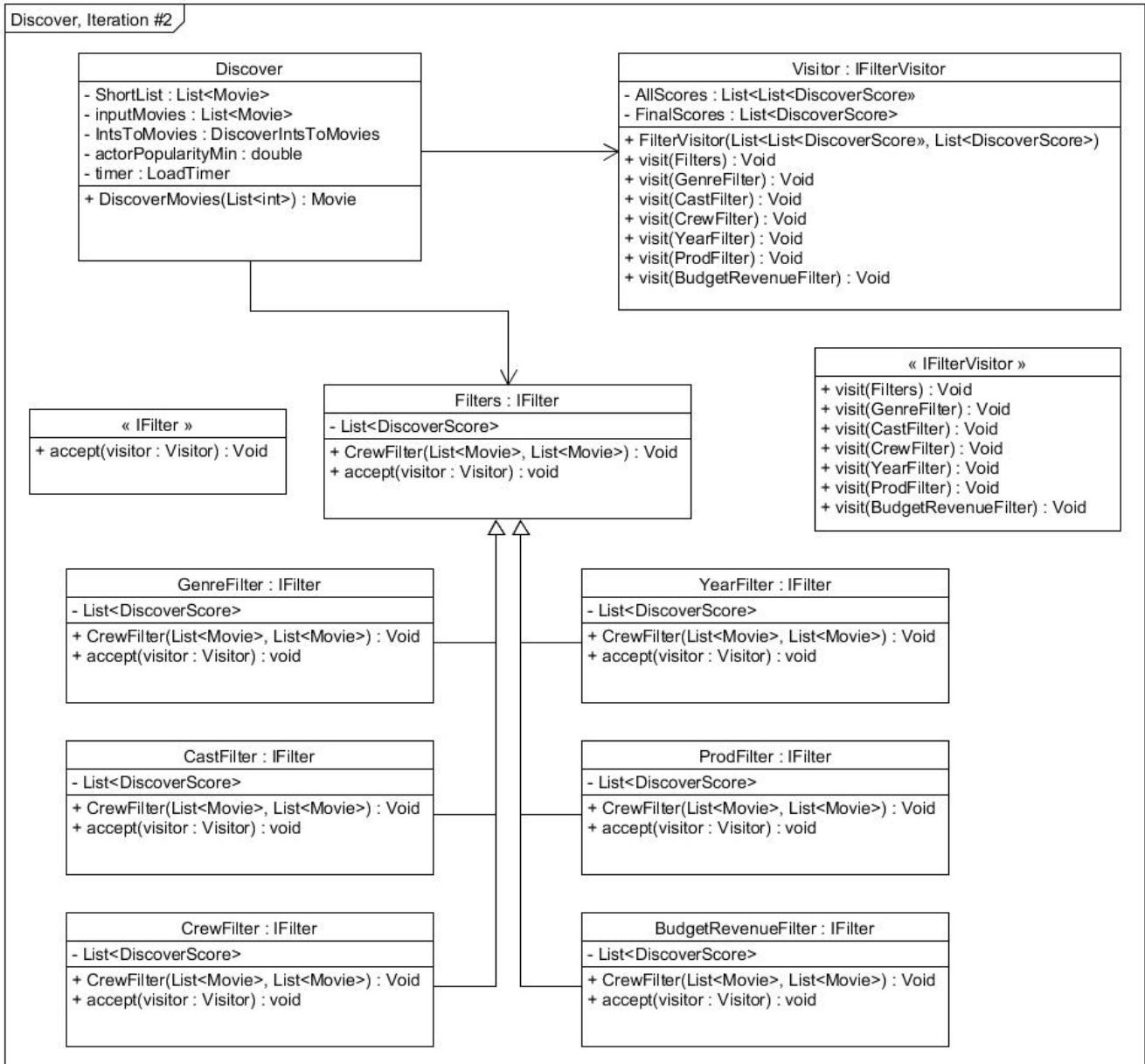


Figure 18: Klassediagram over Discover-backend

I diagrammet kan man se et mere eller mindre almindeligt Visitor Design Pattern, ud over at selve strukturen som en visitor skal besøge ikke er kompleks, som ellers ofte kan observeres i brugen af Visitor Pattern. Den primære argumentation for der bruges Visitor Pattern er at det tillader at de forskellige filtre nemt kan tilføjes el. fjernes, og at der kan tilføjes flere Visitors.

En anden måde at se på brugen af Visitor Pattern, er at følge med i sekvensen af handlinger, ved brug af et sekvensdiagram. Se figur 19.

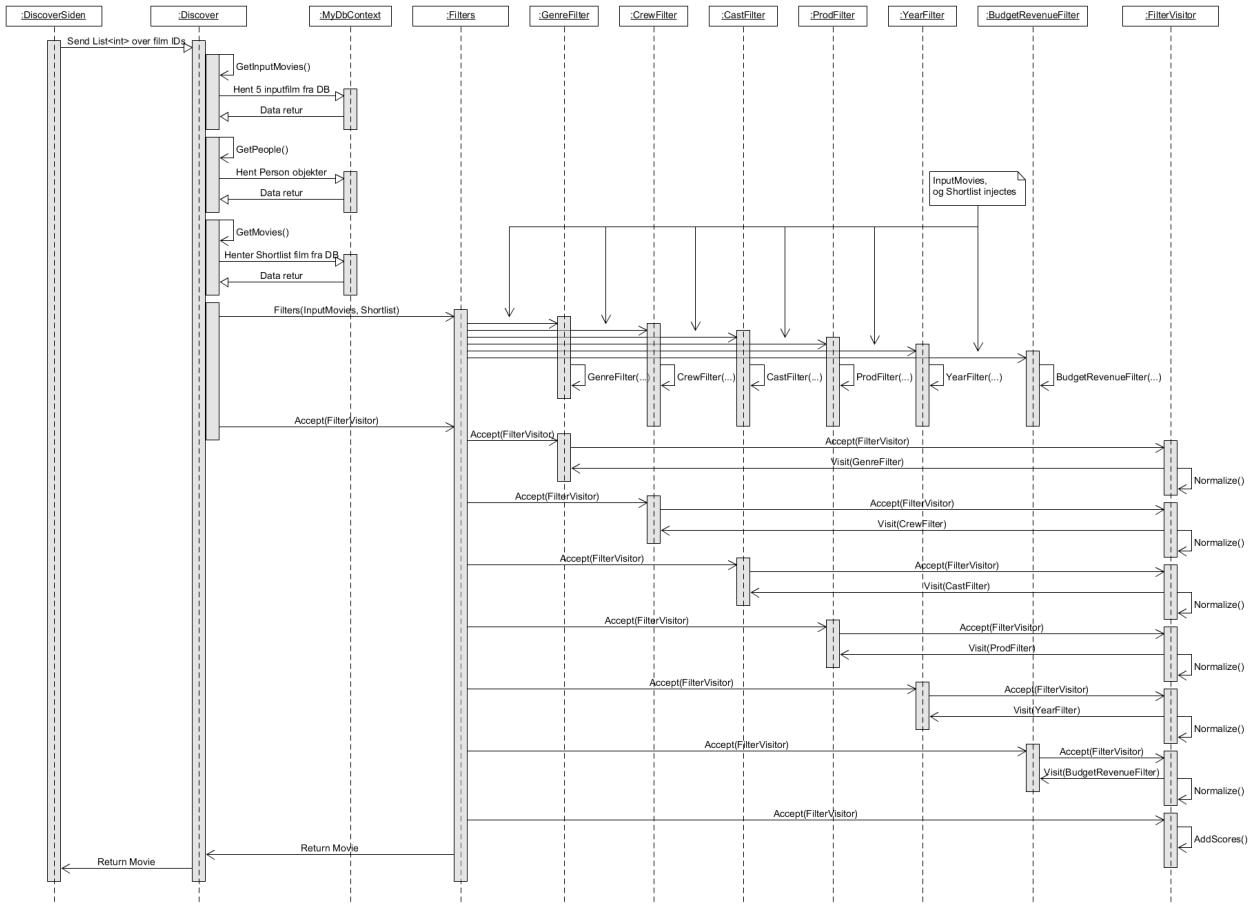


Figure 19: Sekvensdiagram over Discover-backend

Her ses det tydeligt hvordan sekvensen af handlinger forløber, først tilgåes databasen, og alt nødvendigt data hentes, hvorefter Filters bliver oprettet, og derefter bliver Visitoren sat i gang med at besøge hvert filter, for til sidst at returner en film til sidst.

9.4 AutoComplete

AutoComplete er en funktionaliteten som bliver benyttet ved Discover- og Catalogue-siden. AutoComplete er opdelt som et JavaScript program program, der er knyttet til frontend, som både laver REST-API kaldet til backend controlleren, men som også sørger for at opdatere foreslagene på brugerens klient. Derudover er der lavet en backend controller, som laver søgninger på databasen, og returnerer maksimalt 10 foreslag, baseret ud fra det givet input.

9.4.1 Implementering af AutoComplete frontend script

JavaScript implementeringen bruger JQuery, til at hjælpe med at håndtere REST-API kaldet til backend, dette ses på figur 20. Dette script benytter jQuery's autocomplete widget, til at simplificere implementeringen. I denne widget bestemmes det datasæt hvorpå autocomplete laves på, ved at sætte source variablen til et REST-API kald til AutoComplete controlleren, der sørger for at returnere en liste af foreslag.

```

3 // On load/Ready
4 $(document).ready(function () {
5
6
7     //console.log("ready!");
8
9
10    // Setup Autocomplete. Uses Post request to perform tag searches on hydrus.
11    $("#search-field").autocomplete({
12        source: function (request, response) {
13            $.ajax({
14                type: 'POST',
15                url: "/api/autocomplete",
16                contentType: 'application/json',
17                data: JSON.stringify(request.term), // API uses strings in and out!
18                success: function (data) {
19                    response(jQuery.parseJSON(data)); // API uses strings in and out!
20                }
21            });
22        },
23        // 3 characters min to trigger
24        minLength: 3,
25        delay: 200,
26        // Visual of list
27        open: function () {
28            $(this).removeClass("ui-corner-all").addClass("ui-corner-top");
29        },
30        close: function () {
31            $(this).removeClass("ui-corner-top").addClass("ui-corner-all");
32        }
33    });
34 });

```

Figure 20: Javascript Jquery kode, til frontend håndtering af autocomplete

9.4.2 Implementering af AutoComplete backend controller

AutoComplete controlleren er implementeret i ASP.NET ligesom resten af systemets BackEnd. Her oprettes der et POST kald, som frontend'en skal kalde, når der ønskes AutoComplete foreslag. I dette request skal der medsendes en streng i dens body, som POST kaldet laver en søgning på, hvilket kan ses på figur 21. Denne query leder efter film i databasen, som indeholder parameter strengen i deres navn. Hver film i databasen der indeholder parameter strengen i sit navn, vil blive tilføjet til en liste der indeholder filmenes titler. Til sidst tjekkes der for om der er flere end 10 elementer i listen, i så fald vil de resterende elementer blive fjernet, så der maksimalt er 10 elementer i listen. Listen bliver så konverteret til JSON og returneres til brugerens klient.

På figur 21

```

using (var db = new MyDbContext())
{
    // Opretter en ny søgning
    var query = (from m in db.Movies

        // Den søgte værdi SKAL matche enten starten af filmtitelen, ELLER starten af et nyt ord i filmtitelen!
        where m._title.StartsWith(value) || m._title.Contains(" " + value)
        //           Skal starte med ^   Nyt ord, så et space ind foran ^
        // Vi laver den nye sammensluttede tabel
        select new
    {
        label = m._title,      // De hedder label og value, fordi at jQuery autocomplete bruger disse!!!
        value = m._title
    }).ToList();

    if(query.Count > 10)
        query = query.GetRange(0, 10);
}

```

Figure 21: AutoComplete backend - søgning på database

9.5 Databasen

I forhold til vores kravspecifikationer var det nødvendigt at lave en database struktur, som kunne indeholde data med titler på film, indeholde data om personer som står for film skabelsen, og kunne indeholde film fra bestemte år og genre. Her skulle der desuden tages hensyn til at data ville komme fra det eksterne TMDBs API. Denne Data skulle så blive lageret i vores egen database. Vores Database schema kan ses på figur 14 fra DB analyse afsnittet.

9.5.1 Implementering af Databasen

Den udviklede Database endte med at blive implementeret ved at bruge EFcore Frameworket. Her benyttes det indbyggede interface: dbcontext. I DB'en blev der lavet en tabel for hver af nødvendige sæt af attributter. Derudover for at lave en fornuftigt struktur i mellem tabellerne for mange til mange forholdende blev der lavet skyggetabellerne Genres, ProducedBy og employments. Disse 3 skyggetabeller gør det muligt at film kan have flere genre, produktionsselskaber og forskellige ansatte. Hvilket endte med følgende liste af tabeller, se figur 22.



Figure 22: db udsnit af person

En beslutning som hjalp med at effektivisere processen at hente data fra vores database, var at lave plads i vores EFcore schema til at hele lister og objekter som var valgfri. Siden en film kan indeholde

enorme mængder af data, i form af personer arbejdet på filmen, og produktionsselskaber mm. Så er det mest hensigtsmæssigt at kun hente selve filmens stam data, hvis det er det eneste som skal bruges, og så hente alt data til filmen, kun når det faktisk skal bruges. Se figur 23, her kan det ses at der er implementeret flere lister.

```
public class Movie
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
    [Required]
    79 references | 0/13 passing | Andreas, 11 days ago | 1 author, 1 change
    public int movieId { get; set; }
    67 references | 0/13 passing | Andreas, 11 days ago | 1 author, 1 change
    public string _title { get; set; }
    0 references | Andreas, 11 days ago | 1 author, 1 change
    public string? _description { get; set; }
    4 references | Andreas, 11 days ago | 1 author, 1 change
    public string? _posterUrl { get; set; }

    9 references | Andreas, 11 days ago | 1 author, 1 change
    public List<Employment> _employmentList { get; set; } = new();
    5 references | Andreas, 11 days ago | 1 author, 1 change
    public List<Genre> _genreList { get; set; } = new();
    2 references | Andreas, 11 days ago | 1 author, 1 change
    public List<ProdCompany> _prodCompanyList { get; set; } = new();

    13 references | Andreas, 11 days ago | 1 author, 1 change
    public DateTime? _releaseDate { get; set; }
    19 references | Andreas, 11 days ago | 1 author, 1 change
    public int? _budget { get; set; }
    4 references | Andreas, 11 days ago | 1 author, 1 change
    public long? _revenue { get; set; }
    1 reference | Andreas, 11 days ago | 1 author, 1 change
    public double? _popularity { get; set; }
    0 references | Andreas, 11 days ago | 1 author, 1 change
    public int? _runtime { get; set; }

    55 references | 0/13 passing | Andreas, 11 days ago | 1 author, 1 change
    public Movie()
    {
    }
}
```

Figure 23: db Movie tabellen

Som udgangspunkt ville disse lister være tomme, men når en LINQ query bliver anvendt med include(), se figur 24, så kan det vælges til, at de er fyldte.

```
using (var db = new MyDbContext())
{
    InputMovies = db.Movies.Where(c => InputMovieIds.Contains(c.movieId))
        .Include(x => x._genreList)
        .Include(y => y._prodCompanyList)
        .Include(z => z._employmentList)
        .ToList();
}
```

Figure 24: LINQ med include()

Det kan så bestemmes som en query bliver afviklet, hvor meget data der er behov for.

9.6 Design af NUnit testsuite

Der har i projektet været fokus på test af selve Discover algoritmen. Ydermere er der lavet test i forhold til frontend modulet Searchbar. Discover algoritmen består af følgende tre dele.

1. Discover klassen
2. Filter operationer(hjælpefunktioner)
3. Utilities

Test af Discover klassen har først og fremmest til formål at teste filtrene i Discover algoritmen. Testene skal sikre at filtrene modtager lister korrekt samt sortere og prioritere filmene.

Test af filter operationer(hjælpeoperationer) består i test af Normalizing af score, tilføjelse af score til hver enkelt film samt retunering af en korrekt score for de respektive film på listen.

Test af Utilities indebærer test af en Timer, som i projektet undersøger hvor lang tid Discover algoritmen bruger på at finde et resultat, når listen af film sorteres og prioriteres af de valgte filtre.

I følgende afsnit under Test af software, vises implementeringen af nogle af disse test for Discover algoritmen samt enkelte test i forhold til Searchbar.

10 Test af software

10.1 Unittest af Discover Algorithm

Discover-algoritmen dækker over alt kode, som ligger i backend, som arbejder mod at finde et filmforslag ud fra de 5 valgte film. Til Discover klassen er der først og fremmest fokuseret på selve filtrene. Et filter tager hver især to lister, bestående af inputfilmene og opretter en DiscoverScores liste, som indeholder de point som filmene bliver tildelt alt efter det specifikke filter. Testene skal sikre at filterne løser deres opgave, og at de modtager og leverer de rigtige lister.

Implementeringen af de enkelte filtre testes, og nedenfor dokumenteres der i forhold til test af Discover' Cast filtret. Nedenfor på figur 25 testes at der kastes en NullException, når Castfilter benyttes med tomme Movie lists. Unit under test er Castfilter og Assert sikrer den forventede exception kaldes.

```
// TEST FOR INPUT, er listerne Null?
[Test]
• | 0 references | Andreas, 1 day ago | 1 author, 1 change
public void TestCastFilterInput()
{
    // ARRANGE
    CastFilter uut;

    // ACT, ASSERT - Vi send to null lister ind, nu skal den brokke sig!
    Assert.Throws<ArgumentNullException>(() => uut = new CastFilter(nullListe, nullListe));
}
```

Figure 25: Screenshot af Test for Discover Exception

I den næste test, dokumenteret herunder I Figur 26, testes Castfilter med de to stubs. I denne test bruges Assert til at sikre at filtret tilføjer film til DiscoverScores listen. Detter bliver verificeret ved at DiscoverScores listen skal være større end 0, da Castfilter' i så fald, ved brug af inputMovies og Shortlist, har produceret et output i forhold til de to input lister.

```
// TEST FOR OUTPUT, har filteret lavet noget?
[Test]
• | 0 references | Andreas, 1 day ago | 1 author, 1 change
public void TestCastFilterOutputPresent()
{
    // ARRANGE, ACT
    CastFilter uut = new CastFilter(inputMovies, Shortlist);

    // ASSERT, Vi ved at inputlisterne(STUBS) SKAL producere resultater, så DiscoverScore listen SKAL være større end 0
    Assert.That(uut.discoverScores.Count, Is.GreaterThan(0));
}
```

Figure 26: Screenshot af Test for Discover Output

I næste test, dokumenteret herunder I figur 27, testes der at Castfilter ikke blot tilføjer film til discoverScores, men der testes også at alle score værdierne er satte. Assert benyttes her til at undersøge at filmene i discoverScores listen ikke har en score værdi på 0, efter Castfilter' er benyttet.

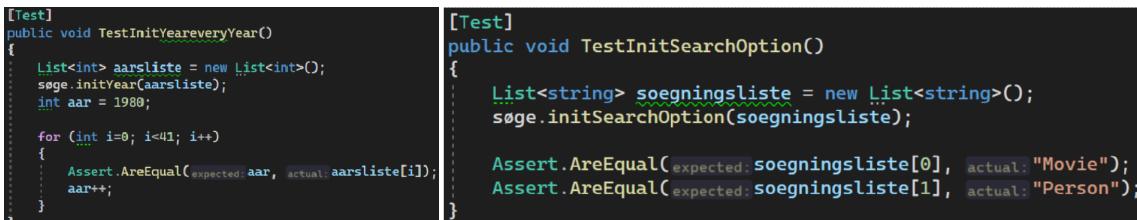
```
// TEST FOR OUTPUT, har filmene fået scores?
[Test]
• | 0 references | Andreas, 1 day ago | 1 author, 1 change
public void TestCastFilterOutputScores()
{
    // ARRANGE, ACT
    CastFilter uut = new CastFilter(inputMovies, Shortlist);

    // ASSERT, Vi ved at inputlisterne(STUBS) SKAL producere resultater, så DiscoverScore listen SKAL kun have scores over null
    Assert.That(uut.discoverScores.Find(x => x.Score == 0), Is.Null);
}
```

Figure 27: Screenshot af Test for Discover Output Score

10.2 Unittest af Search

Der er blevet testet flere elementer af Searchbar bla. dropdown menuerne, som kan ses på figur 28, men der er en grundlæggende udfordring, på trods af refaktorering i 3. iteration af search. Dette kan ses i koden for initGenre(), der laver et direkte DBkald, som har været svært for os at teste med testFrameworket Nunit. Yderligere tests om test af searchbar kan ses i bilagsdokumentationen A.10



```
[Test]
public void TestInitYeareveryYear()
{
    List<int> aarsliste = new List<int>();
    søger.initYear(aarsliste);
    int aar = 1988;

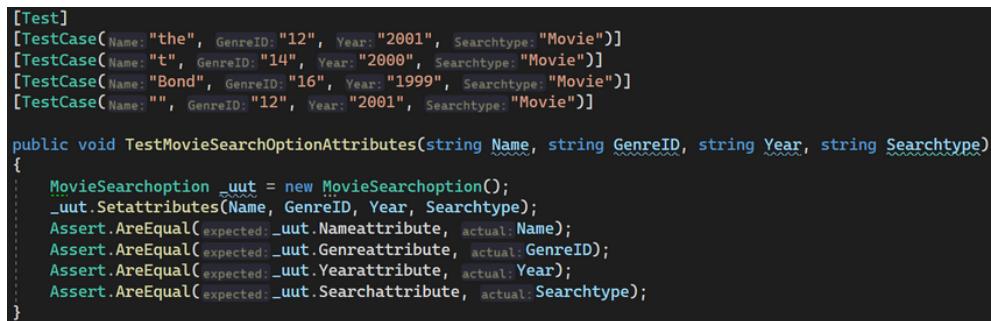
    for (int i=0; i<=41; i++)
    {
        Assert.AreEqual(expected: aar, actual: aarsliste[i]);
        aar++;
    }
}

[Test]
public void TestInitSearchOption()
{
    List<string> soegningsliste = new List<string>();
    søger.initSearchOption(soegningsliste);

    Assert.AreEqual(expected: soegningsliste[0], actual: "Movie");
    Assert.AreEqual(expected: soegningsliste[1], actual: "Person");
}
```

Figure 28: Screenshot af Test for dropdown menuer

For at teste input parametrene var det nødvendigt, at lave metoden Setattributes, samt en public get metode for hver attribute. Dette sikrede, at det kunne testes med forskellige kombinationer af parametre. Se figur 29 med et udsnit af testkoden fra MovieSearchoption().



```
[Test]
[TestCase(Name: "the", GenreID: "12", Year: "2001", Searchtype: "Movie")]
[TestCase(Name: "t", GenreID: "14", Year: "2000", Searchtype: "Movie")]
[TestCase(Name: "Bond", GenreID: "16", Year: "1999", Searchtype: "Movie")]
[TestCase(Name: "", GenreID: "12", Year: "2001", Searchtype: "Movie")]

public void TestMovieSearchOptionAttributes(string Name, string GenreID, string Year, string Searchtype)
{
    MovieSearchoption _uut = new MovieSearchoption();
    _uut.Setattributes(Name, GenreID, Year, Searchtype);
    Assert.AreEqual(expected: _uut.Nameattribute, actual: Name);
    Assert.AreEqual(expected: _uut.Genreattribute, actual: GenreID);
    Assert.AreEqual(expected: _uut.Yearattribute, actual: Year);
    Assert.AreEqual(expected: _uut.Searchattribute, actual: Searchtype);
}
```

Figure 29: Screenshot af Test for SetAttributes()

Searchinput() metoden laver et DBkald, som returnerer en liste af film baseret på søgeparametre. Problematikken i denne klasser er, at den både har ansvar for at modtage data og samtidig filtrere dataen direkte i query kaldet. Klassen kunne være opdelt i to klasser, sa en fik alt data, og anden klasse så filtrede dataen. Dette virkede i overkanten, for at teste en klassens metode. En anden måde så skulle man inject DBcontext klassen, og derefter bruge testframeworket Xunit, men grundet at viden og undervisning er minimal indenfor dette testframework, synes vi at det var et for stort skridt at tage. Helt alternativt skulle der blive oprettet en falsk database, og udskiftet connection strengene. Derfor blev disse metoder ikke testet yderligere, da tid og viden var knap uddover i modultests, som kan læses nedenfor.

10.3 Modultest

10.3.1 Search bar

Searchbar er på vores Catalogue page. Denne sørger for at give en liste af film baseret ud fra brugerens input. Hvis at der gerne læses om yderligere modultest dette læses i bilags afsnittene searchbarmodultest på bilagsnummer A.5.

Her tjekkes dataen fra en brugers søgning op imod TMDBs data. Dette lader sig gøre, ved at dataen fra TMDB ligger i vores egen database struktur, samtidig med at vi filtrer på hvad brugeren får præsenteret på forsiden på baggrund af denne søgning.

På figur 30 vises et udsnit af forsiden med Catalogue page. Her vises der, at OnGet() metoden med at max på 200 film, fungerer som den skal, samt at der er opsat en søgebar, med tilhørende knapper og dropdown menuer.

Welcome To Discover Movies application!

Discover Movies have provided an application, which you can use to find a specific movie, by using the search field and options down below.

You can also try our Discover movie function by pressing the Discover tab. Here you'll provide our algorithm with five movies of your choice, and our algorithm will provide you with a qualified guess of the sixth movie you'll like. This is based on the movies you provided to the algorithm.

Searchfield

For Search Select SearchOption

Search Option

Search result count: 200

Figure 30: Screenshot af Catalogue page

Efterfølgende kigges på hver af dropdown menuer, for at tjekke initialiseringen for om disse er korrekte opsatte. Dette kan ses på figur 31.

Searchfield

For Search Select SearchOption

Search Option

Search Option

Searchfield

For Search Select SearchOption

Select genre

Adventure
Fantasy
Animation
Drama
Horror
Action
Comedy

Searchfield

For Search Select SearchOption

Select Year of creation

Submit

Figure 31: Screenshot valg søgeparametre testet

Autocomplete funktionaliteten tjekkes, og fungerer som den skal. Dette kan ses på figur 32, hvor Autocomplete forsøger at afslutte hvad brugeren vil skrive. Hertil kan det ses, at den kun skal komme med maksimalt 10 resultater.

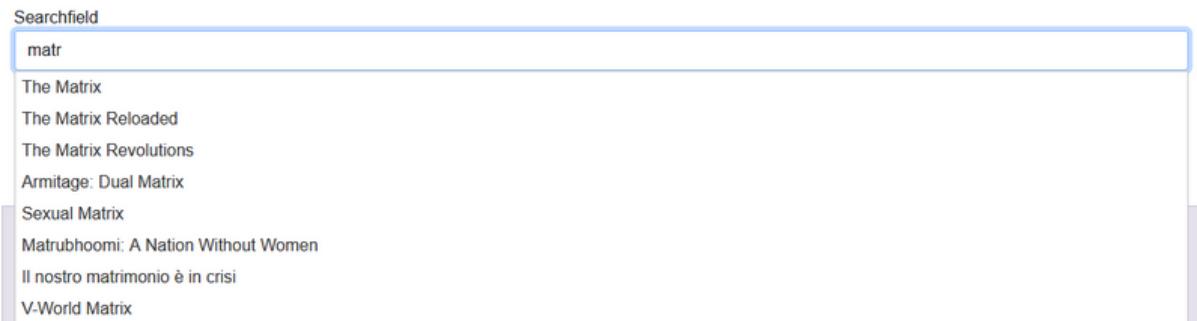


Figure 32: Screenshot Autocomplete

Umiddelbart ud fra søgningen (se figur 33) kan det ses, at der kommer 11 resultater frem ved at skrive "matr".

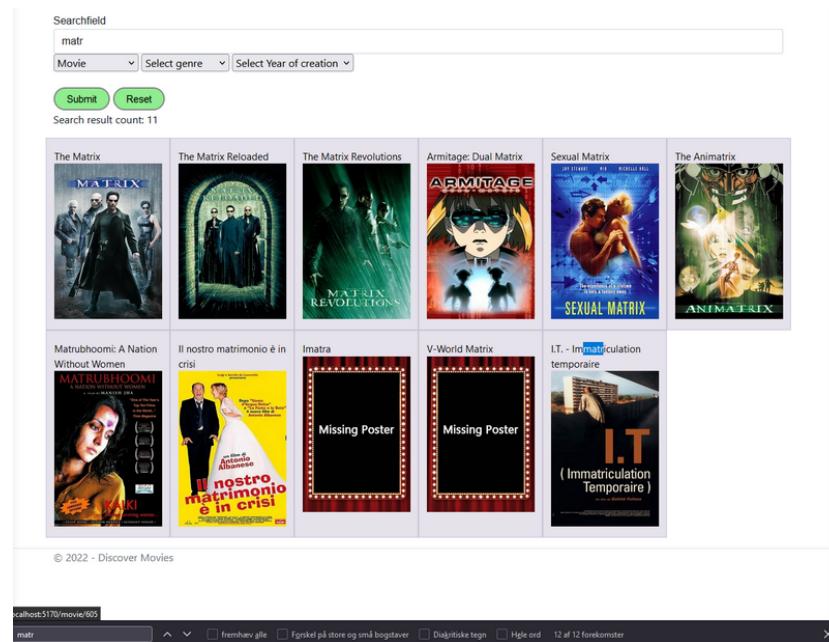


Figure 33: Screenshot af resultater

Der er lavet følgende tabel på figur 34 med flere resultater for forskellige søgeord. Dette er for at give et udsnit af at andre søgninger også virker.

Søgeord	Antal resultater	Hvilke resultater
"on the m"	14	<ol style="list-style-type: none"> 1. "Man on the moon" 2. "Castaway on the Moon" 3. "Brian Regan: I walked on the moon" 4. "Stranded: I've come from a plane that crashed on the mountains", 5. "A Walk on The Moon" 6. "Inuyasha the Movie 4: Fire on the Mystic Island" 7. "Survival on the Mountain" 8. "Conspiracy Theory: Did We Land on the Moon?" 9. "Terminator: Salvation the Machinima Series" 10. "Miracle on the Mountain: The Kincaid Family Story" 11. "Gentle Ben 2: Danger on the Mountain" 12. "Dragon the Master 2" 13. "The Man on the Moon" 14. "A View on The Mountains"
"flying s"	4	<ol style="list-style-type: none"> 1. "Flying Saucer Rock 'N' Roll" 2. "The Flying Scotsman" 3. "Flying Saucers Are Real Volume 2" 4. "FLYING SAUCER 2"
"the dark k"	4	<ol style="list-style-type: none"> 1. The Dark Knight 2. Bloodsport: The Dark Kumite 3. Legends of the Dark Knight: The History of Batman 4. Shadows of the Bat: The Cinematic Saga of the Dark Knight – Reinventing a Hero

Figure 34: Tabel af soegningsresultater modultest

Derfor kan det konkluderes, at denne test fungerer som den skal, både ud fra det gennemgået eksempel, og figuren med tabellen. Hertil kan der lægges mærke til ved søgning af "matr" på figur 33, at missing poster er blevet implementeret i frontenden af catalogue page, da der før ikke blev loadet et billede.

Herefter testes de andre parametre som genre og år. Ud fra en test lignende den tidligere "matr" test, så søges der nu i stedet på genren "adventure". Hertil bliver vi nødt til at kigge på den tidlige data, og den nye data for at konkludere om dette er korrekt. Se figur 35.

Searchfield
matr
Movie Adventure Select Year of creation
Submit Reset
Search result count: 3

The Matrix Reloaded The Matrix Revolutions Armitage: Dual Matrix

Figure 35: Screenshot af Soegningsresultat af matr med genre

For at give et indblik i hvordan der er blevet testet imod forventet resultater ud fra brugerens input parametre, gives her et eksempel På en test med brug af themoviedb.org. På søgningen af "matr" og

adventure kom, "the Matrix reloaded" frem. Ved at slå denne film op på themoviedb.org (TMDB). kan vi netop se, at eventyr (adventure) er en del af deres genreliste. Se figur 36.



Figure 36: Screenshot af Soegningsresultat af Armitage på TMDB

For at gå tilbage og tjekke, om der var nogle af de andre foreslæde film som var adventure, trykkes der på dropdown menuen igen, for at vælge "select genre". Dette betyder at genren nu ikke længere er en del af søgningen. "The Matrix" genre tjekkes fra figur 33, som ikke var en del af søgningen med adventure. På figur 37, Her kan det ses, at adventure ikke er en del af genrelisten på "The Matrix".



Figure 37: Screenshot af Soegningsresultat af Matrix på TMDB

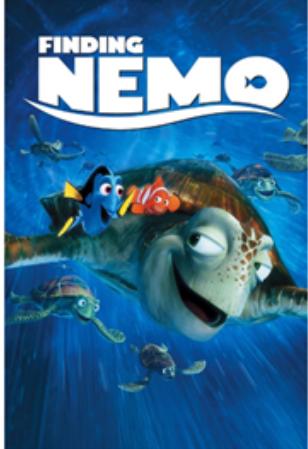
Dermed kan det konkluderes, at The Matrix ikke kom frem, da vi søgte på Searchoption Movie, med "matr" i søgerfeltet, og Genren Adventure, da det IKKE er en adventure film. Samme fremgangsmetode er lavet for flere parametre. Se nedenstående figur med en tabel over nogle eksempler. Se figur 38. Yderligere modultests og forklaringer kan ses i bilagene om modultest af Searchbar.

Searchfield	Searchoption	Genre	Year	Antal Resultater	Hvilke resultater
"the red"	"Movie"	Thriller	0	3	1. The Red Violin 2. The Redemption: Kickboxer 5 3. The Red Siren
"the red"	"Movie"	0	2001	1	1. Rudolph the Red-Nosed Reindeer & the Island of Misfit Toys
"blue"	"Movie"	Drama	2003	2	1. The Blue Light 2. Old, New, Borrowed and blue
"blue"	"Movie"	Animation	2000	4	1. Blue's Big Musical Movie 2. Refrain Blue: Chapter 1 - Scarlet Remembrance 3. Refrain Blue: Chapter 2 - Beneath the Moon... 4. Refrain Blue: Chapter 3 - Eternal Blue Waves
"blue"	"Movie"	0	2000	6	1. Blue's Big Musical Movie 2. A Fighter's Blues 3. Refrain Blue: Chapter 1 - Scarlet Remembrance 4. Refrain Blue: Chapter 2 - Beneath the Moon... 5. Refrain Blue: Chapter 3 - Eternal Blue Waves 6. The Making of Lunar 2: Eternal Blue Complete

Figure 38: tabel med forskellige parametre i søgning tmdb

10.3.2 Movie page

Discover Movies Home Discover Privacy



Title: Finding Nemo

Runtime: 100 minutes

Release: 30/05/2003

Overview:
Nemo, an adventurous young clownfish, is unexpectedly taken from his Great Barrier Reef home to a dentist's office aquarium. It's up to his worrisome father Marlin and a friendly but forgetful fish Dory to bring Nemo home -- meeting vegetarian sharks, surfer dude turtles, hypnotic jellyfish, hungry seagulls, and more along the way.

Genres:
Animation
Family

Cast & Crew:

- Willem Dafoe - Acting - Gill (voice)
- Laura Marano - Acting - Additional Voices (voice)
- Vanessa Marano - Acting - Additional Voices (voice)
- Stephen Root - Acting - Bubbles (voice)
- Jessie Flower - Acting - Additional Voices (voice)
- Bruce Spence - Acting - Chun (voice)
- Eric Bana - Acting - Anchor (voice)
- Albert Brooks - Acting - Marlin (voice)
- Allison Janney - Acting - Peach (voice)
- Brad Garrett - Acting - Bloat (voice)
- Mickie McGowan - Acting - Additional Voices (voice)

Budget/Box Office Revenue: 94000000 / 940335536

© 2022 - Discover Movies

Figure 39: Billede af Movie page, Finding Nemo

På vores enkelte filmside ses det at de informationer vi gerne ville have vist faktisk dukker op, for at tjekke om disse informationer passer sættes de op imod informationer fra wikipedia. Det ses at Box

office revenue passer tilnærmelsesvist, med den information tmdb har oplyst. Det ses at skuespillere (stemme) såsom Willem Dafoe, og Albert Brooks dukker op på listen over medspillere. Udgivelses datoens stemmer overens med datoens på wikipedia.



Figure 40: Billedet af Wikipedia, Finding Nemo

10.3.3 Discover

Introduktion:

Denne modultest omhandler discover funktionaliteten, fra både front og backend. Dette gøres ved at gennemføre en søgning, og tage et kig på resultaterne. Alle testene er lavet ud fra de samme fem film:

- Star Wars: The Clone Wars, Starship Troopers
- Starship Troopers
- Star Trek: first contact
- The Lord of the Rings: The Return of the King
- Harry Potter and the Philosopher's Stone

Målet med testene er at teste at ved input af fem film, skal frontenden vise et filmforeslag til brugeren, og backenden skal vise pointfordelingen for hvert filter, og den samlede pointfordeling.

Frontend

Den første del af testen er front end siden. Her skal der udvælges fem film for at discover algoritmen går i gang. På figur 41 er der en igangværende søgning, vi kan se søgefeltet hvor der indskrives titlen på den ønskede film, og nedenfor en liste af de allerede-valgte film. Begge dele af testene er lavet med de samme film. Der er her valgt en række forskellige film med forskellige elementer tilfældes.

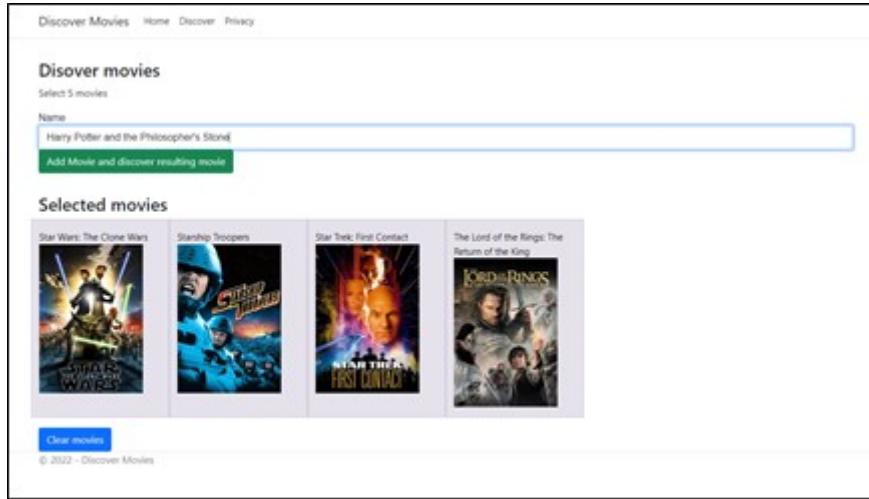


Figure 41: Screenshot fra Discover Pagen under søgning, med filmplakater fra TMDB

På figur 42 kan vi se resultatet af søgningen. Vi kan se de valgte film, en knap til at fjerne dem for at starte en ny forslagssøgning, og nederst på siden den resulterende film som algoritmen har fundet. I dette tilfælde er resultatet den anden film i ringenes herre trilogien. Denne del af testen fungerer som en stor black box, men da vi har valgt den tredje af ringenes herre film, kan vi se at der er en sammenhæng der, der vil strække sig over flere af filtrene.

Da testen forløb som forventet, og vi fik et forslag tilbage ud fra inputtet, kan vi konkludere at vores discover page fungerer. Kvaliteten af forslaget er svært at bedømme, men som tidligere nævnt, kan vi se at der er en tydelig sammenhæng med mindst en af filmene.

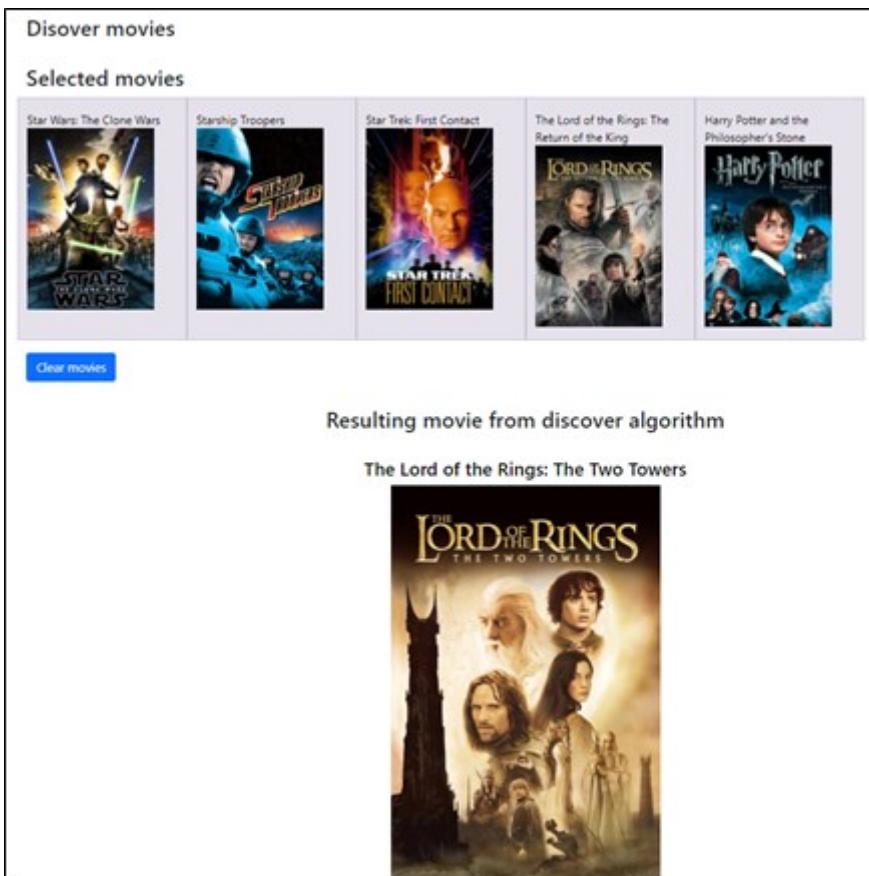


Figure 42: Screenshot fra Discover Pagens Resultat, med filmplakater fra TMDB

Backend

For at kunne teste backenden er der blevet sat op til at kunne vise resultater for de individuelle filtre, og det samlede resultat, efter normaliseringen. Det er valgt kun at vise de 10 film med flest point for hvert filter og det samlede resultat da listerne ellers bliver for lange at vise, og man ville ikke kunne tyde meget da der kommer flere og flere film, som pointtallene bliver lavere.

Genrefilter

Genrefiltret tildeler point efter genrer. Resultaterne på figur 43 viser de 10 højest scorende film. Vi kan se at sci-fi og fantasy begge er genrer der går igen. Action går også igen igennem flere af filmene. Dette stemmer godt overens med de film vi gav til algoritmen, hvor fantasy, sci-fi og action går igen. Vi kan også se at der er flere forskellige mængder point givet ud, så vi kan se at filtret tildeler pointene forskelligt.

FILTER: GENRE	

Monkeybone	- 15
Star Trek: Insurrection	- 14
Star Trek: Nemesis	- 14
The Matrix Reloaded	- 14
The Matrix Revolutions	- 14
The Island	- 14
Fantastic Four	- 14
Thunderbirds	- 14
Street Fighter: The Legend of Chun-Li	- 14
Fantastic Four: Rise of the Silver Surfer	- 13

Figure 43: Resultat for genrer

Samlet resultat

For det samlede resultat ser vi på figur 44 en tydelig vægtning af film. Mange af filmene ser vi score højt i flere affiltrene, og flere af top filmene er dele af serier og franchises af film der var inkluderet i de fem givne film. Det første tal for hver film er den score filtrene sammenlagt har givet den efter normaliseringen. Det andet tal er den popularitet TMDB har givet den. Vi kan se på resultatet at den højest scorende film er den anden film i ringenes herre trilogien, det er det samme resultat som vi ser i testen af frontenden, der har brugt de samme inputfilm. Vi kan se at vi får en række forskellige scores, dette viser os at filtrene har givet filmene forskellige værdier som ønsket.

Vi kan på baggrund af dette, og de andre filtre, konkludere at vores filtre samlet kan udvælge den film der gennemsnitteligt passer bedst med de fem film der bliver givet til algoritmen, baseret på de parametre vi arbejder med. Vi kan til sidst også konkludere at det er den samme film vi får i algoritmen, som den der bliver vist på frontenden. Testene for de sidste fem filtre kan findes i bilag A.7 DiscoverModulTest.

```

Final scores:
-----
FinalScores: 221
The Lord of the Rings: The Two Towers: 4,90 - 154,461
The Lord of the Rings: The Fellowship of the Ring: 4,56 - 118,641
Harry Potter and the Chamber of Secrets: 3,73 - 282,119
Star Trek: Insurrection: 3,65 - 23,303
Harry Potter and the Prisoner of Azkaban: 3,51 - 234,743
Star Trek: Nemesis: 3,33 - 27,934
Fantastic Four: Rise of the Silver Surfer: 3,26 - 36,373
Fantastic Four: 3,22 - 43,37
Face/Off: 3,03 - 40,245
Harry Potter and the Order of the Phoenix: 3,02 - 212,585
Time taken: 00:00:16.6691541

```

Figure 44: Samlet og resultatet efter normalisering

10.4 Acquire

For at modulteste dette program ser vi på de krav der bliver stillet, som er at downloade filmdata og lægge det i databasen, uden fejl, og at programmet skal kunne startes og stoppes, uden tab af data, og at det skal være i stand til at downloade et helt årti værd af filmdata.

Test	Resultat
Modultest 1: Bliver downloadede filmdata gemt korrekt?	Passed
Modultest 2: Kan Acquire startes og stoppes uden databab?	Passed
Modultest 3: Har databasen mindst et årtis-værd af film	Passed

Ved at køre programmet periodisk gennem hele udviklingen, nåede databasen op på over 33.000+ film, som dækker over et interval af 15 år. Dette giver Discover de bedste chancer for at give relevante forslag.

10.5 Integrationstest

Siden samtlige features i systemet er blevet udviklet sideløbende, så har hvert merge af en feature til projektets main git branch givet andledning til en integrationstest. Hver gang er det så både blevet testet manuelt. NUnit tests har også spillet en væsentlig rolle i integration, siden flere tests er komplette integrationstest, dette er dog særskilt for Discover-forslagsfunktionen, hvor der er to komplette integrationstests i form af NUnit tests. Her bruges kæmpe stubs, som er dumps fra databasen, der injectes som objekter, som stedfortræder for database klassen, og som så kan vise at det forventede resultat kommer ud i den anden ende. Testen kan ses i figur 45.

```

[Test]
● 0 references | Andreas, 6 days ago | 1 author, 2 changes
public void TestDiscoverOutput()
{
    // ARRANGE
    DiscoverMoviesProduction.Discover uut = new DiscoverMoviesProduction.Discover();
    var MockDiscoverDB = new Mock<IDiscoverDB>();
    var MockDiscoverIntsToMovies = new Mock<IDiscoverInputMovies>();

    Console.WriteLine("Person count: " + persons.Count);
    Console.WriteLine("ShortList count: " + Shortlist.Count);
    List<int> movieInts = new List<int> { 199, 1571, 1639, 1893, 2787 };

    // Mock setup m. DB stubs
    MockDiscoverDB.Setup(x => x.GetPeople(It.IsAny<List<int>>()).Returns(persons);
    MockDiscoverDB.Setup(z => z.GetMovies(It.IsAny<List<int>>()).Returns(Shortlist);

    MockDiscoverIntsToMovies.Setup(y => y.GetInputMovies(movieInts)).Returns(inputMovies);

    // ACT
    Movie newMovie = uut.DiscoverMovies(movieInts, MockDiscoverDB.Object, MockDiscoverIntsToMovies.Object);

    // ASSERT – Den skal vende tilbage med en film!
    Assert.That(newMovie, Is.Not.Null);
}

```

Figure 45: Screenshot af den primære NUnit integrationstest

10.6 Accepttest

Herunder er opskrevet de tests der blev udført under opsyn af Lars, i tableform, testne blev udført udfra kravspecifikationen, hvor kun de krav der står i krav kolonnen var udearbejdet. Testene som blev udført er opskrevet i test kolonnen og resultatet af disse tests i resultat kolonnen, De sidste to krav, altså 8 og 9, var krav ment til anden iteration af produktet, men med lidt tid i overskud endte disse med at blive udviklet, og derfor også testet.

Nummer	Krav	Test	Resultat
1	Skal indeholde database, database skal indeholde titel, folk der står bag film (instruktør, skuespillere osv.). Database skal indeholde film fra et bestemt årti.	Databasen bliver undersøgt igennem Azure Data Studio for film i et bestemt årti.	OK
2	Skal indeholde GUI, der skal tilgås gennem hjemmeside.	Hjemmesiden bliver besøgt, det ses at der er et GUI til rådighed.	OK
3	Skal indeholde backend, backend skal have en algoritme, der laver crosscorrelation mellem valgte film og kommer med filmforslag.	Hjemmesidens Discover funktion bliver anvendt, det ses at der fås et filmforslag. Desuden kan yderligere information findes i konsollen.	OK
4	Databasen skal klones fra en ekstern database.	Acquire programmet kores.	OK
5	GUI skal indeholde et katalog af filmtiteler, som der kan søges i.	Forsiden af hjemmesiden browses igennem.	OK
6	GUI skal have en side med dynamiske filmforslag ud fra de 5 valgte film	Der navigeres til hjemmesidens Discover funktion.	OK
7	Hver film skal have egen side/model.	Der klikkes ind på en enkelt film fra hjemmesidens forside/katalog.	OK
8	GUI burde inddrage API til coverbilleder til alle film	Der klikkes ind på en enkelt film fra hjemmesidens forside/katalog. Det ses at der er opstillet et coverbillede af den givne film.	OK
9	GUI burde inddrage API til filmbeskrivelser	Der klikkes ind på en enkelt film fra hjemmesidens forside/katalog. Det ses at der er opstillet en beskrivelse af den givne film.	OK

11 Perspektivering

11.1 Systemarkitektur

Systemarkitekturen, der kan ses på figur 46, viser at systemets moduler generelt set har høj kobling med SQL Database da alle controllere, moduler samt backend til pages kalder på databasen. Der er på figuren, figur 46 udvalgt 2 koblinger der umiddelbart kan undgås.

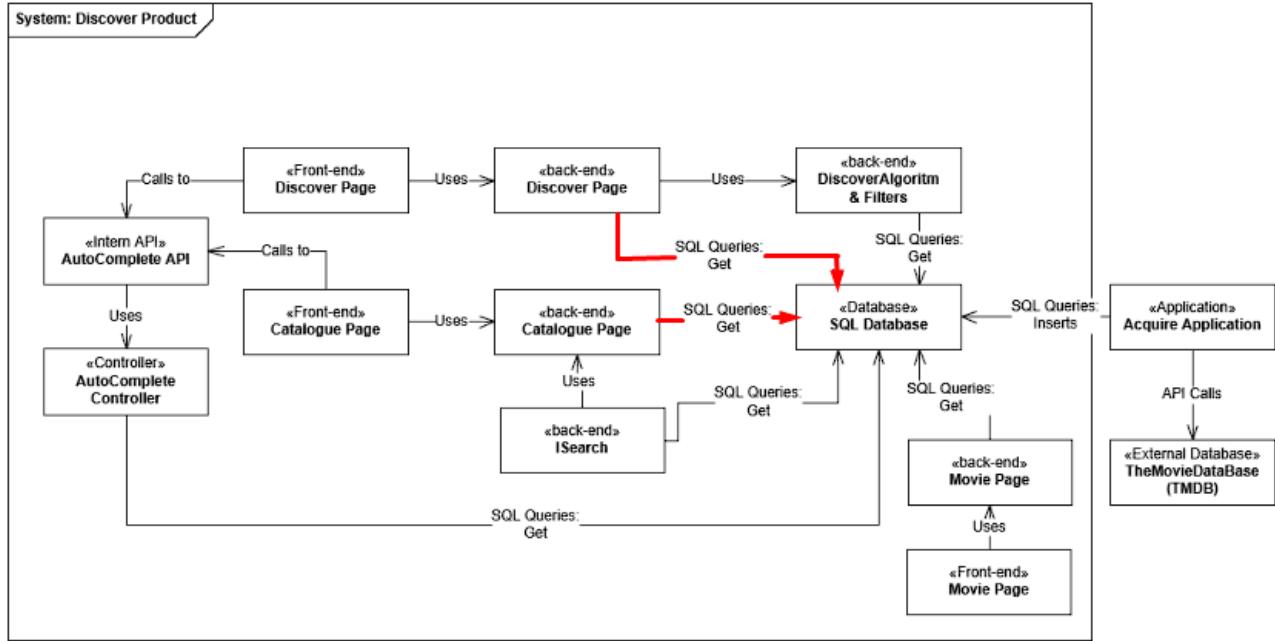


Figure 46: Systemarkitektur, med markeringer ved queries fra Razor Page backend til SQL Database

Catalogue Page kalder på databasen under OnGet, altså ved initialiseringen af siden. Databasen bruges også ved OnPost, altså når en bruger søger. Under søgninger og under alle fremtidige indlæsninger af film-kataloget, bliver ResolveSearch brugt som et lag af abstraktion til backenden og database queries.

ResolveSearch kunne altså også anvendes under OnGet til at indlæse den første visning af kataloget, og dermed sænke koblingen i systemet. Searchbaren kunne være implementeret som en dymisk søgebar, som på IMDB.com. Denne kan detektere hvad en bruger søger efter. Dette ville dog kræve yderligere iteragration mellem DB og catalogue page, samtidig med at dele af algoritmen

Discover Page kalder også både direkte til databasen, og anvender DiscoverAlgorithm der også tilgår SQL Database. Dette er mindre problematisk, da der stor forskel på de queries der kaldes i Discover Page backend og de queries der kaldes i DiscoverAlgorithm.

For at opretholde SRP(Single Responsibility Principle), ville der i fremtidige pages laves et lag af abstraktion således pages ikke direkte querier databasen.

11.2 Discover-Algoritmen

En vigtig opførsel af forslagsfunktionen, er at den skal indsamle data om de personer som har hhv. arbejdet foran og bag kameraet på de valgte input film. Denne opførsel kan dog også gå hen og blive problematisk som databasen vokser, og især alt efter hvilke film som bliver valgt. Det er fordi der ingen grænser er sat ind i søgningen, som der søges på personer, og shortlisten genereres. f.eks. Søges der på Steven Spielberg, og ses der kun på instruerede film, så er der ca. 32 film, men ses der på en instruktør som Takashi Miike, så er der altså langt over hundrede film som skal indlæses. Det ses også

tydeligt i koden, hvor Where kriteriet bliver flittigt brugt, se figur 47.

```
Console.WriteLine("Henter alle skuespillere involveret:");
// Ny liste over personer, som skal bruges til at finde alle VIP fra de fem inputfilm
people = (from pers in db.Persons Where(x => x._Personpopularity > 20)
          join emp in db.Employments Where(x => inputMovieInts.Contains(x._movieId))
          on pers._personId equals emp._personId
          where emp._job == "Acting"
          select pers).ToList();

Console.WriteLine("Henter alle instruktørere og Producere involveret:");
people.AddRange((from pers in db.Persons
                  join emp in db.Employments Where(x => inputMovieInts.Contains(x._movieId))
                  on pers._personId equals emp._personId
                  where emp._job == "Director" || emp._job == "Producer"
                  select pers).ToList());
```

Figure 47: Databasekaldet for at skaffe personers data

En måde at komme uden om dette problem ville være at begrænse enten hvor mange film der må være på en givet shortlist, el. begrænse hvor mange personer der må findes film fra, ud fra inputfilmene. F.eks. der kunne bruges .Take(x) funktionen i selve søgningen, hvor x kunne være grænsen.

11.3 Continuous Integration

Under udviklingen og implementeringen af Discover-Forslags klasserne, blev det først prioriteret ret sent i processen at der skulle laves tests løbende. Det blev nedprioriteret af flere årsager, bla. en stram tidsplan, for at komme i mål med den basale funktionalitet af systemet, men også fordi at der netop ikke dukkede mange problemer op under selve implementeringen, så manglen på tests var ikke åbenlys.

Da det så blev tid til at se på tests, efter den første iteration af funktionaliteten, blev fordelene så meget åbenlyse, da det både hjalp med at styre refaktoringen af koden, men også at det skulle takes højde for ekstremer i de værdier som passerer rundt mellem klasserne. Som tests så dækkede mere og mere af koden, kunne fordelene ses langt tydeligere. F.eks. Så blev det meget tydeligt at injection og brug af interfaces giver en masse fordele, og som tests blev lavet for at teste værdier, så blev det også tydeligt at arbejdet i sig selv, med at lave tests, kan minde en om hvilke senarerier at koden skal forberedes til. Det er bestemt noget at tage med til det næste projekt, at starte CI tidligt, hvis det skal gavne, og hvis det skal hjælpe til at skabe bedre kode.

11.4 Statisk database

Siden at databasen, brugt i Discover, er downloadet fra TMDB, og der ingen forventninger er til at data i Discover's database skal opdateres, så kan det præsentere et problem senere i systemets levetid. Det er fordi at filmdata i TMDBs egne database ændre sig over tid, eksempelvis Popularity attributten er en der ændres, som film og personer løbende stiger eller falder i popularitet. Dette afspejles derfor ikke i Discover's Database. En anden problematik kan være hvis en person skifter navn, så ville dette navneskifte ikke figurere i vores database, hvis personen allerede er lagt i databasen. En måde at komme uden om det på, er enten at løbende hente nye popularity værdier via deres API, el. hente alt data direkte fra deres database, dog med den risiko at deres side muligvis ikke kommer til at eksistere for evigt.

11.5 FrontEnd

Valget om at skrive frontenden som razor pages, blev taget tidligt i semesteret, på et tidspunkt hvor det var det eneste web baseret frontend, som gruppen havde fået undervisning i. Skulle gruppen tage dette valg igen, på nuværende tidspunkt, havde det højest sandsynligt ikke have været det samme

valg. Frontenden endte med at blive udviklet meget lig en SPA (Single page application), hvilke er meget nemmere og renere at lave med et bibliotek såsom React, som vi har fået undervisning i senere på semesteret. De elementer af vores side der matcher en SPA, er f.eks. menu baren i toppen som i alle tilfælde er den samme, den er implementeret, i vores system, ved at have et shared layout til alle siderne, hvilket kunne laves bedre i React uden at genbruge den samme mængde af kode.

12 Konklusion

Arbejdet på projektet, DiscoverMovies, er endt med et funktionelt produkt, der opfylder kravene for et minimal viable product. Produktets software design er opsat således, at videreudvikling på produktet er ligetil.

Der er lavet en testsuite med grundige unit tests af softwaren, samt foretaget en lang række modultest for at sikre at systemet er stabilt, og at alle fejl eller mangler er dokumenteret.

Overvejelser omkring godt design har været gennemgående og kommer til udtryk både i system- og softwarearkitekturen, og især ved valget og refaktorering af DiscoverAlgorithm til at følge visitor pattern.

DiscoverMovies fungerer både som et katalog, således data om film kan findes, men systemets primære funktionalitet er at løse choice paralysis ved valg af film. DiscoverMovies giver filmforslag der er uafhængige af specifikke streamingstjenesters potentielle agendaer, såsom optimering af seer-tid eller promovering af ”originals” produceret internt. Forslagene tager heller ikke udgangspunkt i en brugers tidlige bagkatalog, men i stedet giver fuld autonomi til brugeren af Discover funktionaliteten til at udvælge 5 film og få forslag udelukkende ud fra disse. Dermed er problemstillingen blevet løst af DiscoverMovies.

Selve udviklingsprocessen har fungeret godt, med en klar struktur for arbejdet imens der blev opretholdt en godt miljø til at stille spørgsmål ved beslutninger og have givende diskussioner og samtaler omkring hvad den korrekte fremgangsmåde for vores gruppe samt produkt ville være. Scrum, med tilhørende agile board, har været et vigtigt og godt værktøj til at planlægge og tracke opgavers status. Derudover havde vi lavede i starten af processen en forventningsafstemning, der hjalp os med at afgrænse projektet og opsætte et solidt minimal viable produkt at arbejde imod. Dette er resulteret i at forholdet mellem hvor meget tid vi har haft til projektet, og den arbejdsbyrde vi har lagt på os selv overordnet stemte overens. Dette er som sagt resulteret i et færdigt produkt, med god dokumentation, og fornuftige designmæssige beslutninger.

13 Personlige Konklusioner

13.1 Oliver

Det har været en fornøjelse, at samarbejde på dette semesterprojekt. Jeg er godt tilfreds med hvordan gruppen har håndteret arbejdsopgaver, da der blev vist engagement fra alle gruppemedlemmer. Dette betød at der var konstant fremskridt i projektarbejdet, men havde også betydning for at holde projektet relevant og spændende gennem semesteret. I gruppen har vi ikke brugt en stram ansvarfordeling, hvor gruppemedlemmer kun har ansvar for et område (eksempelvis database, backend eller frontend), i stedet har vi haft en mere flydende fordeling af ansvar, som har sikret at alle i gruppen har haft indflydelse på de diverse elementer af produktet. Denne strategi har bestemt haft sine fordele - og har været god at prøve, selvom jeg nok stadigvæk bryder mere om en strammere opdeling.

13.2 Anders

Generelt synes jeg projektet er gået rigtig fint. Vi er kommet i mål med alle de krav vi tidligt i processen stillede til produktet og har holdt en fornuftig arbejdsindsats gennem hele processen. Der har været få problemer med at den undervisning vi kunne have brugte til udviklingen af produktet først blev introduceret senere i processen (såsom react), men dette er detaljer i et ellers fint foreløbet semesterprojekt. Gruppedynamikken fungerede også super fint og en grundig forventningsafstemning i starten af processen sørgede også for at vi kom igennem projektet uden interne frustreringer.

13.3 Patrick

Projektarbejde og samarbejde i dette semesterprojekt har fungeret rigtig godt. Det har givet mig et stort udbytte, at vi benyttede de samme teknologier, som vi lærte i dette semester modsat andre semestre. I starten var jeg nervøs for at arbejde på et rent software projekt, heldigvis synes jeg det er gået godt, og det har været meget spændende kun at arbejde med software folk. Det at projektrammerne har været meget frie, har givet mig stor inspiration til at kunne bruge tekniske begreber til at løse nogle af vores konkrete problemstillinger i projektet. I vores semesterprojekt har der været stor fokus på scrum metoder og scrumboards. Det har mig givet ro og overblik over projektet. Desværre opdagede vi forsent i projektet glæden ved CI, og software test, ved at det blev nedprioriteret. Dette vil jeg have fokus på at lave tidligere til næste gang.

13.4 Mads

Semesterprojektets udførelse er gået godt. Det har været tydeligt fra starten at vi alle har haft en masse erfaringer med, der har gjort udførelsen af projektet mere organiseret. Dette har især kunnet ses ved rapportskrivningen, der er forgået i et mindre hektisk tempo end ved tidligere semesterprojekter, og i at vores minimum viable product, var skåret ind til benet, med rum til videreudvikling hvis der var tid. Vores brug af scrum, og den arbejdsfordeling der tilhører har også været en stor hjælp til at holde en organiseret arbejdsgang igennem projektet. Dette har sørget for at man vidste hvad der skulle laves, og at der ikke er blevet glemt opgaver. Igennem projektet har jeg haft rig mulighed for at udnytte teknikker fra det nuværende og tidligere semestre. Det skete dog også at teknikker blev givet til os for sent i semestret til at benytte dem, og det har begrænset mulighederne, eksempler på dette er react og mongoDb. Til det næste projekt jeg skal arbejde på vil jeg især ligge vægt på succesen med brug af scrum, og dens agile udviklingsmetode.

13.5 Rasmus

Gennemførslen af dette semesterprojekt er gået godt og uden de store problemer, både fagligt samt i forbindelse med gruppearbejdet. Projektet har været spændende at arbejde med, indeholdt interessante faglige elementer fra semesterets andre kurser og haft en fornuftig størrelse således arbejdsbyrden har været fornuftig. Da projektet er kørt ved siden af de andre kurser, har det været tilfældet, eksempelvis med databasen, at ny viden fra et kursus gør at dele i projektet skulle laves om. Det har

medført nogen grad af spildt arbejde, som kunne være undværet, og i andre tilfælde at de ting vi lærer senere på semesteret, ikke kan nå at være med i projektet som eksempelvis React. Måden på hvor vi har arbejdet i små grupper(2-3 personer) om enkelte dele har personligt fungeret godt for mig. Det har været en anden måde at arbejde på end fra tidligere semesterprojekter og en god personlig læring.

13.6 Casper

Med klare idéer om mulige produkter fra Andreas, startede projektet med en forespørgsel om hvorledes disse var noget resten af gruppen havde lyst til at arbejde med. Da det blev fastlagt i gruppen, at det var nogen gode muligheder tog vi et valg, og endte med at arbejde med en filmforslag. Der har generelt i gruppen været et godt samarbejde, med god uddelegering af arbejde, som i denne omgang blev gjort igennem redmine. Med ny erfaring i brug af redmine har jeg bestemt at det ikke er noget jeg personligt vil anvende igen. Det blev valgt at skrive rapport dokumentet i latex, med yderligere dokumentation gemt i et onedrive, hvilket er en måde at skrive projekt på, jeg selv havde prøvet før, og var glad for.

13.7 Andreas

Det har været en spændende proces at gennemføre dette projekt, både fordi det har været et rent-software projekt, modsat tidligere semesterprojekter, men også fordi det omhandler så mange web teknologier, som alle interessere mig. Vi har både arbejdet med databaser, kommunikation med APIer, lavet vores egne APIer osv. En udfordring som opstod mod forventning, var at som vi arbejdede sammen med git, trods bedste intentioner og planlægning, formåede vi alligevel at få problemer undervejs, som vi mergede vores features sammen. Jeg vil bestemt stræbe for at forbedre processen næste gang jeg skal sammenarbejde med andre med git som værktøj. Jeg er også rigtig glad for slutresultatet ifht. forslagsalgoritmen, siden jeg har haft meget med den at gøre, og fordi det er en problemstilling som har ligget mig nær i en del år.

A Bilag

Bilag indeholder en oversigt over projektets vedlagte bilagsstruktur. Bilag refereres til gennem navn, og kan findes i ”Bilag.zip” ved at følge den angivne sti.

A.1 Tidsplan

En tidlig tidsplan over hvornår de forskellige sprint skulle forløbe.

Sti: Bilag/Tidsplan.

A.2 Mødedokumenter

Alle mødedokumenter gennem processen, inklusiv dagsordner samt referater.

Sti: Bilag/Mødedokumenter”

A.3 Redmine Logbog

Logbog eksporteret direkte fra Redmine baseret på tasks.

Sti: Bilag/Logbog

A.4 Dokumentation af Searchbar

Dokumentation af softwaren bag search bar.

Sti: Bilag/DokumentationAfSoftware/SearchBar

A.5 Dokumentation af Searchbar Modultest

Dokumentation af modul test af search bar.

Sti: Bilag/DokumentationAfSoftware/SearchbarModultest

A.6 Dokumentation af DiscoverAlgorithm

Dokumentation af softwaren bag DiscoverAlgorithm.

Sti: Bilag/DokumentationAfSoftware/Discover/DiscoverAlgorithm

A.7 Dokumentation af DiscoverModulTest

Dokumentation af modultest af discover page samt discover algoritmen.

Sti: Bilag/DokumentationAfSoftware/Discover/DiscoverModulTest

A.8 Dokumentation af Front-End

Dokumentation af den samlede front-end samt tidlige skitser.

Sti: Bilag/DokumentationAfSoftware/SamletFrontEnd

A.9 Dokumentation af Movie Page

Dokumentation af modultests for Movie Page.

Sti: Bilag/DokumentationAfSoftware/MoviePage

A.10 Dokumentation af NUnit

Dokumentation af den samlede NUnit testsuite.

Sti: Bilag/DokumentationAfSoftware/NUnit

A.11 Dokumentation af Database

Dokumentation af Databasen, med logical schemas.

Sti: Bilag/DokumentationAfSoftware/Database

A.12 DiscoverMovies

Selve DiscoverMovies applikationen som en visual studio solution. Sti: Bilag/DiscoverMovies

References

- [1] Pinal Dave. *SQL Terms vs MongoDB Terms*. 2020. URL: <https://blog.sqlauthority.com/2020/04/18/sql-terms-vs-mongodb-terms/>.
- [2] José Gabriel Navarro. *Box office market share of Disney in the United States and Canada from 2000 to 2021*. 2022. URL: <https://www.statista.com/statistics/187300/box-office-market-share-of-disney-in-north-america-since-2000/>.
- [3] José Gabriel Navarro. *Box office market share of Universal in the United States and Canada from 2000 to 2021*. 2022. URL: <https://www.statista.com/statistics/187307/box-office-market-share-of-universal-in-north-america-since-2000/>.
- [4] Anupam Pareek. *Netflix case study — Breaking paradox of choice*. 2019. URL: <https://uxplanet.org/breaking-paradox-of-choice-netflix-case-study-7f29107d1e2b>.