University of Groningen

Web Engineering

Project Report

Authors:
Rares Dobre
(s4051203)
Adrian Segura Lorente
(s3913007)
Daniel Skala
(s3953602)
Felix Zailskas
(s3918270)
Daniel Baxan
(s4066480)

Lecturer:
dr. V. Andrikopoulos

January 10, 2021

Contents

1 Introduction

We have designed an Web application which allows the user to manipulate and view data from the IMDB movie dataset. Our application allows the user to use diverse tools which allow filtering of specific movies, actors, genres and also provide statistics for the movies. The user is able also able to manipulate data by adding, updating and deleting movies from the system.

2 Application implementation

For the development of this application we have decided to go with the MEVN (MongoDB, Express, Vue, Node) stack. We use MongoDB as a noSQL database since it allows us to manipulate collections of entities, which in our case are the movie, actors and genres.

For the back-end framework we use Express because of it's accessible functionality. For the front-end part our decision was to use the Vue.js framework, since it allows a simple implementation of all the required functionality in comparison with Angular and React technologies. Since we are using java script for the back-end we also used the Node.js as the runtime environment and the npm (node packet manager) for handling all the node-modules that were required in the development/production version of the program.

The communication between the client and the server is done via the concept of the REST API. This means that the server sends the user only the requested data and then the client side handles that information and displays it in a proper way. A further explanation about the models, API requests and UI implementation will be discussed in the following sections.

2.1 User requirements

The following requirements were made for the application:
⋆ It is worth to mention that the preparation of the data objects will be made on the back-end side of the web application and the front-end will deal only with the display of the obtained objects.

1. Retrieve all actors in the dataset, optionally filtered by (full) name;

   In the case the user does not want to filter this is done with a GET request from the client to the server of the form /api/actors.
   Otherwise we use the the GET request of the form /api/actors/{*name*}.

2. Retrieve all available information about a specific movie identified by its unique IMDB URL or by its (non-unique) title;

This is done with a GET request from the client to the server of the form /api/movies/{URL} in the case if the user wants to identify the movie by URL.
In the case the user wants to find the movie by it's title then the GET request of form /api/movies/{$title$}.

3. Retrieve all movies by a specific actor or director identified by name and/or in specific year

   This is done with a GET request from the client to the server of the form /api/movies/{$name$} with the corresponding query parameters.

4. Retrieve all movie genres for a specific actor or director, optionally sorted by year (ascending or descending)

   This is done with a GET request from the client to the server of the form /api/genres/{$name$} with the corresponding query parameters. For the ascending/descending ordering we will have an additional query parameter which will indicate in which ordering the application should return the result. The User will be able to select a specific order by a using drop-down menu.

5. Retrieve an ordering of the movies ranked by their popularity (user rating) from more to less popular, with the possibility to subset this order;

   This is done via a GET request from the client to the server of the form /api/movies/.

   To subset them, the desired specifications limit will be passed through query parameters.

6. Retrieve an ordering of movies in a specific year ranked and subsetted by popularity as above;

   This is done with a GET request from the client to the server of the form /api/movies/.

   To specify the desired year and the subset, query parameters will be used.

7. descriptive statistics (mean, median, standard deviation) for the popularity of all movies for a particular actor with an optional filter by year

   This is done by a GET request from the client to the server of the form /api/statistics/{$name$}.

3 Application technologies

This section covers the different technologies used to implement the restful API, together with a description for each element and the given usage in the API.

Name	Description	Type	Usage in API
Node.js	Runtime environment that executes javascript code on the serverside	Runtime environment	Used across the whole backend of the API
MongoDB	NoSQL database system used	Database System	Used to store the data used by the API
Express	Backend web application framework for Node.js	Framework and Package	Used in the implementation of the API in general to speed up development
Mongoose	Object data modeling library for MongoDB and Node.js	Library	Used to perform queries to the NoSQL database
CORS	HTTP-header based mechanism that allows the server to indicate any other origins than itself	Package	Used in the API to allow the indication of other origins outside of the server
body-parser	Parsing middleware used to parse request bodies, facilitating the access to the request and its information	Package	Used extensively to access the body of requests
nanoid	Compact unique string identifiers generator	Package	Used to generate identifiers for newly created movies for the database
Quickchart	Chart image generator	External API	Used to generate charts from the statistics generated in our API
Papa-parse	JSON to CSV and CSV to JSON converter	Package	Used to convert json content to csv from queries
Axios	Promise based HTTP client for the browser and node.js	Package	Used to send requests to third party API
Morgan	HTTP request logger middleware for node.js	Package	Used to display HTTP request log messages in the terminal
Bootstrap-vue	A framework for building responsive websites	Framework	Used in UI components in frontend
Material Design for Bootstrap	UI KIT for building responsive websites	Plugin	Used in the html report

4 Model

This section describes all the model entities we have used during the development of our application.

Movies	A movie in the database
Title	Name of the movie
Rating	Rating of the movie
Year	Release year of the movie
User Rating	Rating of the movie based on the user
Votes	Number of votes
Metascore	Metascore of the movie
IMG URL	URL of the image of the movie
Countries	Countries in which the movie was released
Languages	Languages in which the movie was released
Actors	Actors involved in the movie
Genre	Genre of the movie
Tagline	Short description of the movie
Description	Description of the movie
Directors	List of directors that directed a movie
Runtime	Length of the movie (in minutes)
IMDB URL	IMDB URL of the movie

Actors	An actor
Name	Full name of the actors
$\wedge$! Movies	Movies the actor stars in

Genres	A genre of a movie
Name	Name of the genre
$\wedge$! Movies	Movies of that genre

5 API Design

Here we represent all the API requests and entities used in the back-end.
It is worth to mention that all GET requests can be sorted and limited by the release year.
The user is free to choose the desired representation of the data. Each API endpoint will support both CSV and JSON representations, and when none is selected, the JSON (default) will be chosen.

5.1 Movies

/api/movies/	GET	request all movies with the possibility to sort them based on popularity and to subset them (either through the quantity, or through the year), which will be provided through query parameters Success: 200 Not found: 404 Server error: 500
/api/movies/url/{*URL*}	GET	request a specific movie by its url Success: 200 Not found: 404 Server error: 500
/api/movies/title/{*title*}	GET	get the movie by title Success: 200 Not found: 404 Server error: 500
/api/movies/name/{*name*}	GET	get the movies by actor's/director's name The role of the corresponding person and the year will be specified through query parameters Success: 200 Not found: 404 Server error: 500
/api/movies/random/	GET	request a number of random movies with the possibility to select the number of movies Success: 200 Not found: 404 Server error: 500
/api/movies/	POST	create a new movie Success: 201 Server error: 500
/api/movies/url/{*URL*}	DELETE	delete a move by it's unique IMDB URL Success: 202 Not found: 404 Server error: 500
/api/movies/url/{*URL*}	PUT	update info above a movie by it's unique IMDB URL Success: 200 Not found: 404 Server error: 500
/api/movies/random/	GET	get random movies to watch today Success: 200 Not found: 404 Server error: 500
/api/movies/rating/{*rating*}	GET	get all movies with specific rating Success: 200 Not found: 404 Server error: 500

5.2 Actors

Request	Type	Description
/api/actors/	GET	request all actors from the server Sucess: 200 Server error: 500
/api/actors/{*name*}	GET	request one or more actors by their name we use string matching in order to get all the valid results from the DB Sucess: 200 Not found: 404 Server error: 500

5.3 Genres

Request	Type	Description
/api/genres/	GET	request all genres from the server Success: 200 Server error: 500
/api/genres/{*name*}	GET	request one or more genres by the name of the actor/ director we use string matching in order to get all the valid results from the DB The role of the corresponding person and the year will be specified through query parameters Sucess: 200 Not found: 404 Server error: 500

5.4 Statistics

Request	Type	Description
/api/statistics/{*name*}	GET	request the statistics of movies of an actor using that actors name The year will be specified through query parameters Sucess: 200 Not found: 404 Server error: 500

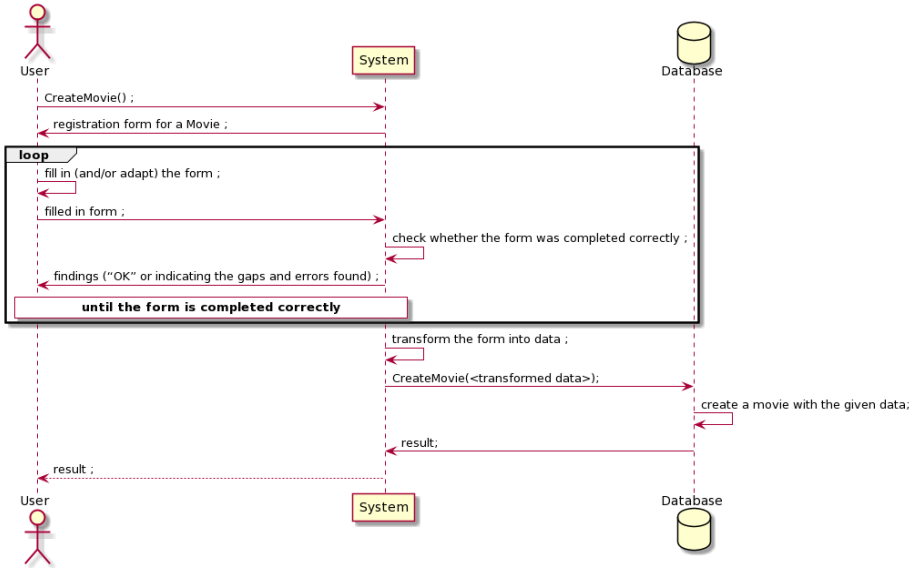
6 Example requests

This sections contains schema representations of request in order to have a better understanding of the application work-flow. Here we discuss an example of each CRUD operation implemented in our program.

6.1 Create a movie

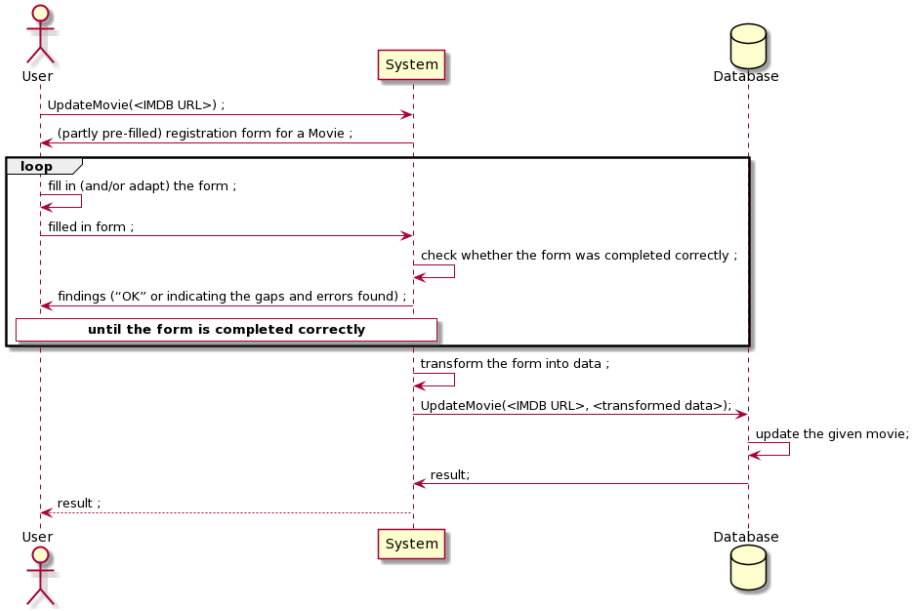
This example describes a use case for creating a movie and adding it to the database. After creation it will be present in the database with all values as-

signed by the user.



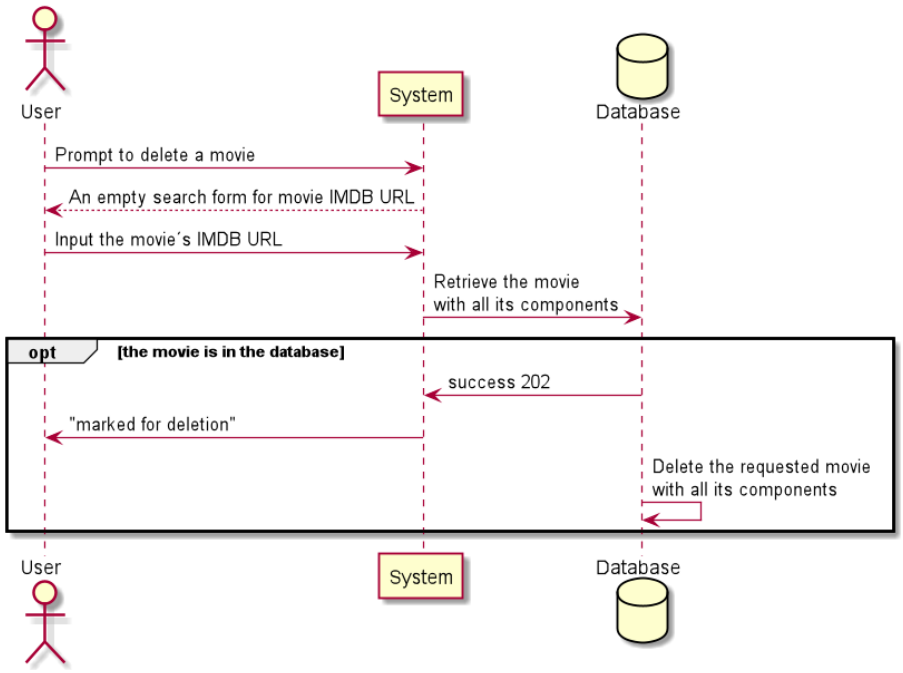
6.2 Update a movie

This example describes a use case for updating data of an existing movie in the Database. After updating, the movie will exist with the same id and all indicated fields will have the updated values. Movie is identified with its unique IMDB URL which is inputted from the user.



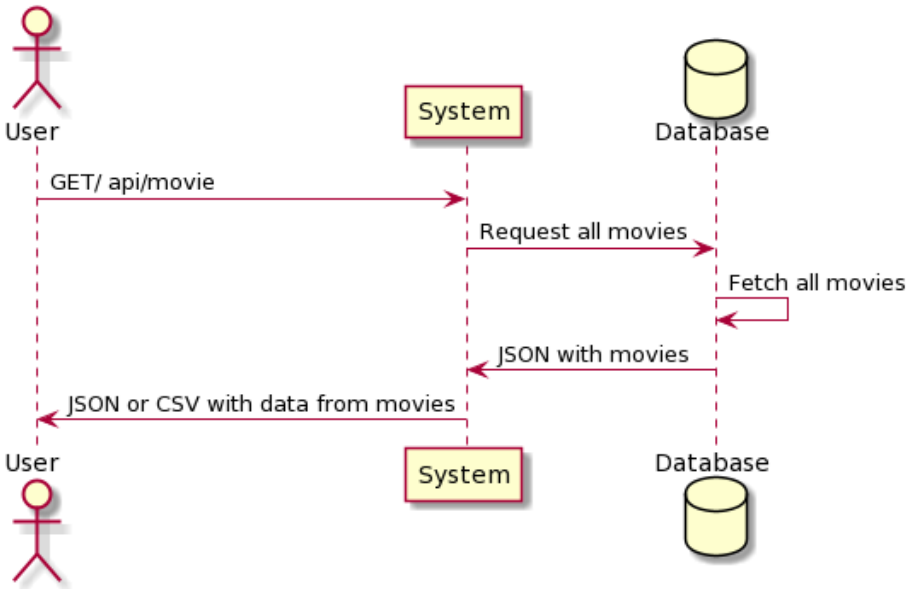
6.3 Delete a movie

This example describes a use case for deleting a specific movie from the Database. After deletion, the movie and all its components will not be present in the database. Movie is identified with its unique IMDB URL which is inputted from the user.



6.4 Retrieve all movies

This example describes the use case for retrieving all movies from the database. The user sends a get request to the server and receives a JSON or CSV representation of all movies stored in the database. The displaying, filtering and sorting of the data is done by the user.



7 Error Handling

7.1 Server errors

In case of an error occurring on the server side (e.g. no connection to the database) the user will receive a status code of 500. Along with the status code, the produced error will be sent to the user.

7.2 Resource not found error

In case the User has requested a resource that is not available on the server (e.g. a movie that does not exist in the database), the server will reply with a status code of 404. Along with the status code, the produced error message will be sent to the user.

8 Collections

This section contains the information about database collections used in our project.

8.1 Movies

- title: *string*
  title: given title for movie

- rating: *string*
  title: rating for movie
- year: *string*
  title: year of release
- users_rating: *string*
  title: rating for movie given by users
- votes: *string*
  title: count of votes given for movie
- metascore: *string*
  title: metascore assigned to movie
- img_url: *string*
  title: poster url
- countries: *[string]*
  title: countries-of-first-Release
- languages: *[string]*
  title: list of languages in which the movie was dubbed
- actors: *[string]*
  title: list of actors involved in the movie
- genres: *[string]*
  title: list of genres of the movie
- imdb_url: *string*
  title: the imdb url corresponding to the movie

9 UI implementation

All the functionality described in this section has been implemented using the Vue.js framework.
The User Interface has a form-like structure to get the input from user and uses UI components to display the output of the requested query. Such components include cards, tables or lists.
The structure of the front-end looks as follows:

1. Drop-down menu to select a query
2. Optional input fields to input the name of the actor or title of a movie
3. Optional radio buttons for sorting or other attributes
4. A checkbox button for a CSV representation of the data
5. Submit Button to send the query

To display the graph of the statistics, we use the link of the generated graph from quickchart.io which we display under the form.

10 Extra functionality

- Sub-string matching for actors names
- Getting all genres query
- Third party API (*quickchart.io*) usage for visualization of statistics (histogram of movie ratings for specific actor)
- Visualisation of statistics
- Additional endpoint for getting a number of random movies to watch today
- Additional endpoint to get all movies with specific rating

11 Milestone outcomes

In this section we describe the feedback we got for each milestone and improvements we made based on that feedback.

11.1 M1 outcome

- We changed the singular form of the nouns that we were using to denote collections to plural and we have updated the routes to reflect it
- We modified the delete use case diagram, returning a 202 code meaning "marked for deletion"
- We added additional specifications on how the ascending and descending order will be specified by the User
- We added additional information for the documentation to clarify the meaning of getting "one or more actors", as well as for genres
- We have changed the endpoints to improve readability

11.2 M2 outcome

- The code duplication issue regarding the csv converter part in each controller was fixed by using a single function for each file.
- The superfluous comments were deleted from self-explanatory components in the code.

11.3 M3 outcome

12 Distribution of work

In this section we describe the general and particular task each member had to do. The contribution to the report was equal for each group member. We were working together on weekly sessions (4-5 hours) using Visual Studio Code Live Share extension so that we could write the code/report in parallel.

13 Deploying of the application

This section describes the set of instructions that a user needs to follow in order to run the current Web application.

1. Install mongodb locally on the platform and start it. For a guide on how to install mongodb Community Edition please visit: **this link**.
2. Download the github repository.
3. Go to: **here** and download the files(the movie.csv file and movie.json folder), then copy thoser inside the local repository on your machine into a new folder called *datasets*.
4. Run "npm install" in an terminal inside the root directory of this repository and inside the */frontend/webeng-app/* folder.
5. To start the app run the command **npm run full** inside the root directory.

14 Changelog

This section contains the changes of the report during the development period.

- **1.0** Changes and data of V 1.0
- **1.1** Updated API requests and added responses, status codes.
  Added error handling section.
  Added query parameters to the requests.
  Updated user requirements implementation.
  Changed PATCH request to PUT.
- **1.2** Plural nouns for collections
  Edited according to the feedback from TAs
- **1.3** Additional extras for milestone 2
  Added technologies table

- **1.4** Additional extra endpoints

- **1.5** Added the outcome for milestone 2